JESÚS PIZARRO PELÁEZ

IES TRINIDAD ARROYO

DPTO. DE ELECTRÓNICA

ÍNDICE

ÍNDICE 1
1. INTRODUCCIÓN A LOS MICROCONTROLADORES 4
1.1. ARQUITECTURAS VON NEUMANN Y HARDVARD 4
1.2. ARQUITECTURAS CISC, RISC Y SISC 5
1.3. APLICACIONES DE LOS MICROCONTROLADORES 6
2. CARACTERÍSTICAS PRINCIPALES DEL PIC16F88 7
2.1. MEMORIA 7
2.2. PERIFÉRICOS 7
2.3. CARACTERÍSTICAS ESPECIALES 8
2.4. PATILLAJE 8
3. ESTRUCTURA INTERNA Y ORGANIZACIÓN DE LA MEMORIA 11
3.1. DIAGRAMA DE BLOQUES 11
3.2. MEMORIA DE INSTRUCCIONES 12
3.3. MEMORIA DE DATOS 13
3.4. MAPA DE MEMORIA 14
3.5. VALORES INICIALES 16
4. JUEGO DE INSTRUCCIONES 17
4.1. INSTRUCCIONES ARITMÉTICAS 18
4.2. INSTRUCCIONES LÓGICAS 19
4.3. INSTRUCCIONES ORIENTADAS A BIT 19
4.4. INSTRUCCIONES DE CARGA Y MOVIMIENTO DE DATOS 20
4.5. INSTRUCCIONES DE CONTROL DE FLUJO (SALTO CONDICIONAL) 20
4.6. INSTRUCCIONES DE SALTO 21
4.7. OTRAS INSTRUCCIONES 21
4.8. RESUMEN DE INSTRUCCIONES 22
5. REGISTROS ESPECIALES 23
5.1. STATUS 23
5.2. OPTION_REG 25
5.3. PCL Y PCLATH 26
5.4. INDF Y FSR 27
5.5. OSCCON 27

5.6. OSCTUNE ... 29
5.7. CONFIG1 ... 29
6. PUERTOS A Y B ... 33
6.1. PORTA Y PORTB ... 33
6.2. ANSEL ... 33
6.3. TRISA Y TRISB ... 34
7. TEMPORIZADORES ... 36
7.1. TEMPORIZADORES Y CONTADORES ... 36
7.2. TIMER0 ... 36
7.3. TIMER1 ... 38
7.4. TIMER2 ... 41
8. MÓDULO CCP ... 43
8.1. PRINCIPIOS DE LA MODULACIÓN PWM ... 43
8.2. CCP1CON ... 45
8.3. MODULACIÓN PWM CON EL PIC16F88 ... 46
9. CONVERSOR A/D ... 48
9.1. SEÑALES ANALÓGICAS Y DIGITALES ... 48
9.2. ANSEL ... 49
9.3. ADCON0 ... 49
9.4. ADCON1 ... 50
9.5. ADRESL, ADRESH ... 51
9.6. PROCESO DE CONVERSIÓN ... 52
10. EEPROM ... 54
10.1. CARACTERÍSTICAS ... 54
10.2. EEDATA Y EEADR. EEDATH Y EEADRH ... 54
10.3. EECON1 Y EECON2 ... 55
10.4. LECTURA DE DATOS ... 56
10.5. ESCRITURA DE DATOS ... 56
11. INTERRUPCIONES ... 58
11.1. NECESIDAD DE LAS INTERRUPCIONES ... 58
11.2. INTERRUPCIONES EN EL PIC16F88 ... 59
11.3. INTCON ... 62
11.4. PIE1, PIE2 ... 62
11.5. PIR1, PIR2 ... 63
12. EL ENSAMBLADOR MPASM™ ... 66
12.1. SINTAXIS GENERAL ... 66
12.2. DIRECTIVAS ... 68
13. Desarrollo y SIMULACIÓN: PROTEUS™ ... 76
14. GRABACIÓN: ICPROG™ ... 76

1. INTRODUCCIÓN A LOS MICROCONTROLADORES

Como se ha visto anteriormente, los sistemas microprogramables se pueden dividir en tres grandes grupos:

- Microprocesadores
- Microcontroladores
- Dispositivos lógicos programables

Si recordamos, definimos el microcontrolador como un sistema microprogramable completo, compuesto de un microprocesador, memoria, puertos de E/S, periféricos, etc. integrado en un solo chip y capaz de realizar una determinada función de control sin necesitar apenas componentes adicionales. Los microcontroladores pueden tener distintos tipos de arquitectura interna. Esto va a influir en la forma en que tratan los datos, la velocidad de proceso, rendimiento, instrucciones disponibles… A continuación se citan una serie de parámetros importantes en la arquitectura interna de un microcontrolador.

1.1. ARQUITECTURAS VON NEUMANN Y HARDVARD

Cuando estudiamos los sistemas basados en microprocesador, vimos que estaban formados por una unidad central de proceso o CPU, que se unía a otros dispositivos como la memoria y los puertos de E/S mediante BUSES. En estos sistemas tenemos una memoria unida a la CPU mediante un bus de datos y un bus de direcciones. La CPU lee de la memoria las líneas de código del programa que está ejecutando mediante el bus de datos, y también usamos el mismo bus y la misma memoria para leer y escribir todos los datos de programa. Esta arquitectura es la más frecuente en los sistemas basados en microprocesador y se llama **Arquitectura Von Neumann**.

Existe otra filosofía en cuanto a la arquitectura de los sistemas microprogramables consistente en usar dos memorias distintas cada una con su propio bus. Así, tenemos una *memoria de instrucciones* donde se almacenan las líneas de código del programa, a la cual se accede solo para leer las instrucciones. Es una memoria de solo lectura. La otra memoria es la *memoria de datos* a la cual se accede cada vez que necesitamos leer o escribir un dato de programa. A este tipo de configuración se le denomina **Arquitectura Hardvard**.

En la *figura 1* se puede apreciar el diseño de ambas arquitecturas. La arquitectura Hardvard tiene la gran ventaja de que el sistema tiene un rendimiento mucho mayor, ya que se puede acceder

simultáneamente a las instrucciones y a los datos al usar buses distintos. Por contra, esto requiere un diseño más complejo y por tanto más caro para controlar el sistema.

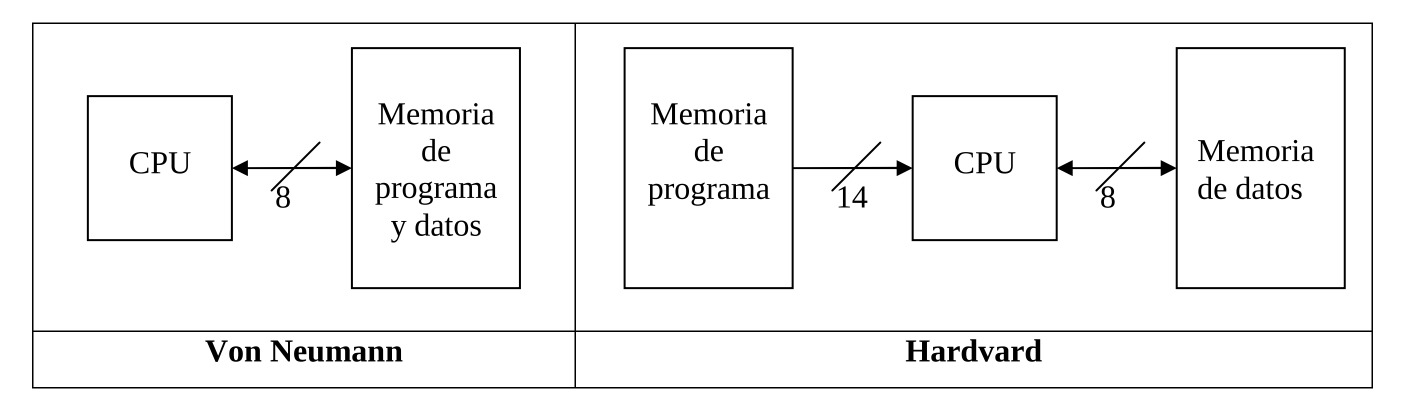


Figura 1: Arquitecturas de CPU

El microcontrolador PIC16F88 usa una arquitectura Hardvard con una memoria de instrucciones unida a la CPU mediante un bus de 14 bits y una memoria de datos unida a la CPU por un bus de datos de 8 bits distinto al anterior.

1.2. ARQUITECTURAS CISC, RISC Y SISC

Hemos visto que el juego de instrucciones de un sistema microprogramable se puede definir como las distintas operaciones que puede realizar dicho sistema. Otra posible clasificación de los sistemas microprogramables es en función de su juego de instrucciones. En este aspecto tenemos tres arquitecturas diferentes:

- CISC (Complex Instruction Set Computer): Computadora con juego de instrucciones complejo. Los sistemas con esta arquitectura tienen muchas instrucciones y pueden hacer varias operaciones distintas. Son más sencillos de programar, pero necesitan un hardware más complejo y caro. Generalmente las instrucciones necesitan varios ciclos máquina para ejecutarse.
- RISC (Reduced Instruction Set Computer): Computadora con juego de instrucciones reducido. Los sistemas con esta arquitectura tienen muy pocas instrucciones que además son muy básicas. Esto hace que sean sistemas con un hardware muy simple, lo que les hace muy sencillos, baratos y rápidos. Como contrapartida, el hecho de poseer pocas instrucciones hace que en ocasiones necesitemos varias instrucciones seguidas para implementar una función que no realiza el hardware.
- SISC (Specific Instruction Set Computer): Computadora con juego de instrucciones específico. Estos sistemas se basan en que su juego de instrucciones es específico para una determinada aplicación (tratamiento de señales, cálculo numérico…). Las instrucciones disponibles dependen de la aplicación del sistema. Son sistemas muy potentes para hacer determinadas cosas, pero no son versátiles al ser el juego de instrucciones tan específico.

El microcontrolador PIC16F88 tiene una arquitectura RISC. Cuenta con un total de 35 instrucciones. Todas ocupan 14 bits y se ejecutan en un único ciclo máquina (4 ciclos de reloj). Las de salto requerirán dos ciclos máquina como veremos más adelante.

1.3. APLICACIONES DE LOS MICROCONTROLADORES

Los microcontroladores son circuitos integrados que se usan para controlar un determinado dispositivo. Esta definición puede parecer bastante amplia y poco estricta, pero es que en realidad es así: tenemos microcontroladores en cualquier sitio al que miremos, haciendo infinidad de labores distintas.

Sin necesidad de salir de casa, podemos encontrar decenas de microcontroladores que, sin que lo sepamos, hacen que todos los aparatos que tenemos funcionen como es debido (lavadoras, relojes, televisores, teléfonos, lectores de MP3, cámaras de fotos, aparatos de radio…) además de todos los sistemas conectados a un ordenador personal, que aunque tenga un microprocesador principal, lleva varios microcontroladores dentro y fuera para control de teclado, ratón, impresora, router... Un automóvil actual lleva varios microcontroladores para controlar desde el encendido hasta la radio, pasando por climatización, luces, frenada, apertura de puertas, regulación de velocidad… En la industria todos los procesos usan gran cantidad de robots… todos ellos controlados por uno o varios microcontroladores. Cuando vamos por la calle, aunque no los veamos están ahí, en un semáforo, una fuente, luces, carteles luminosos, máquinas dispensadoras, cajas registradoras, lectores de códigos de barras… en fin, que podríamos dedicar un documento únicamente a las aplicaciones de los microcontroladores en el mundo actual.

Hay infinidad de microcontroladores en el mercado, todos con sus ventajas e inconvenientes. Nosotros vamos a centrarnos en uno concreto: El **PIC16F88** de la casa **Microchip**. ¿Por qué éste y no otro? En primer lugar porque los microcontroladores PIC son muy usados, por lo que, ya puestos a aprender uno, mejor que pertenezca a una familia tan extendida y conocida como la de los PIC, de tal forma que podamos encontrar información, documentación, programas y ayuda sin excesiva dificultad. ¿Por qué el modelo 16F88? Porque dentro de la gama media es de lo mejor que hay. Presenta, como ya veremos, un gran número de periféricos que lo hacen ser apto para muchísimas aplicaciones, oscilador integrado, modo bajo consumo, etc. Sin olvidar que todo esto está disponible en una sola pastilla de 18 pines por un precio aproximado de 4€, lo que hace que podamos permitirnos plantearnos un diseño y poder llevarlo a cabo de una forma sencilla y por muy poco dinero.

2. CARACTERÍSTICAS PRINCIPALES DEL PIC16F88

2.1. MEMORIA

El microcontrolador PIC16F88 es un microcontrolador con arquitectura Hardvard, lo que significa que usa dos memorias independientes para instrucciones y para datos, cada una de ellas con su propio bus. Todas las instrucciones tienen una longitud de palabra de 14bits, que es el ancho del bus de instrucciones. Además, todas las instrucciones se ejecutan en un ciclo máquina consistente en 4 ciclos de reloj, a excepción de las de salto que necesitan dos ciclos máquina. La memoria de datos es SRAM y la longitud de palabra es de 8bits. Los registros se encuentran mapeados en la RAM. También tiene incluida una memoria EEPROM para almacenar aquellos datos que sea necesario conservar ante una pérdida de alimentación. En la *tabla 1* se puede ver la memoria del dispositivo.

Memoria	Nº bits	Cantidad	Tipo
Instrucciones	14	4096 palabras de 14bits	FLASH
Datos	8	368 bytes	SRAM
EEPROM	8	256 bytes	EEPROM

Tabla 1: Memoria del PIC16F88

La arquitectura de todos los PIC es RISC, por lo que tienen un juego de instrucciones reducido. Este modelo concretamente, tiene un total de 35 instrucciones que se verán en un apartado siguiente. Como avance, decir que todas las instrucciones llevan el operando implícito en la palabra de 14 bits, de tal forma que solo se accede una vez por instrucción a la memoria de programa, al contrario que en otros procesadores que hemos visto como el 65C02, en el cual teníamos instrucciones de 1, 2 y hasta 3 bytes.

2.2. PERIFÉRICOS

El PIC16F88 tiene incluidos una serie de periféricos que le dan gran versatilidad a la hora de utilizarlo para cualquier aplicación de control o comunicaciones. A continuación se citan los periféricos incluidos en el integrado.

- 2 puertos de E/S digital (TTL) A y B de 8 patillas cada uno. Cada una de las patillas puede ser configurada independientemente. El puerto B se puede programar para Pull-Up interno.
- 1 módulo Capture, Compare, PWM (CCP). Con 16bit para captura y comparación y 10bit para PWM.

- ADC (conversor analógico – digital) de 7 canales y 10bits.
- Puerto serie síncrono (SSP) con SPI™ (Master/Slave) e I2C™ (Slave).
- Módulo AUSART/SCI con detección de direcciones de 9bits. RS232 con oscilador interno.
- Dos comparadores analógicos.
- Tres temporizadores.

2.3. CARACTERÍSTICAS ESPECIALES

Además de los periféricos, el PIC 16F88 tiene una serie de características adicionales muy interesantes:

- Velocidad de reloj de hasta 20MHz.
- Bajo consumo: En modo Sleep el consumo se puede reducir hasta 200nW.
- WatchDog timer (perro guardián): Es un contador interno que avisa si el sistema se cuelga para poder reiniciarlo desde software con normalidad.
- Oscilador interno configurable: Una de las mejores novedades. Permite prescindir de la señal de reloj externa y utilizar las patillas para otro propósito gracias a su oscilador interno con 8 frecuencias distintas configurables por software de 31.25KHz a 8MHz.
- 100.000 ciclos de borrado/escritura de la memoria FLASH de programa.
- 1 Millón de ciclos de borrado/escritura de la EEPROM de datos. Tiempo de retención de datos superior a 40 años.
- Depuración y programación serie en circuito (ICSP™).

2.4. PATILLAJE

El PIC16F88 se presenta con varios tipos de encapsulado. Éstos se pueden apreciar en la *figura 2*. Si nos fijamos, el número de patillas es muy pequeño en comparación con todos los periféricos y características de que dispone el circuito. Evidentemente, para que esto sea posible, cada patilla sirve para más de una función, teniendo que seleccionar la que nos interese mediante software en nuestro programa como ya veremos.

Nosotros usaremos el encapsulado PDIP de 18 patillas para tener un fácil montaje en placa protoboard. A continuación se incluye el patillaje del integrado en la *figura 3*, así como una descripción de cada uno de los pines en la *tabla 2*.

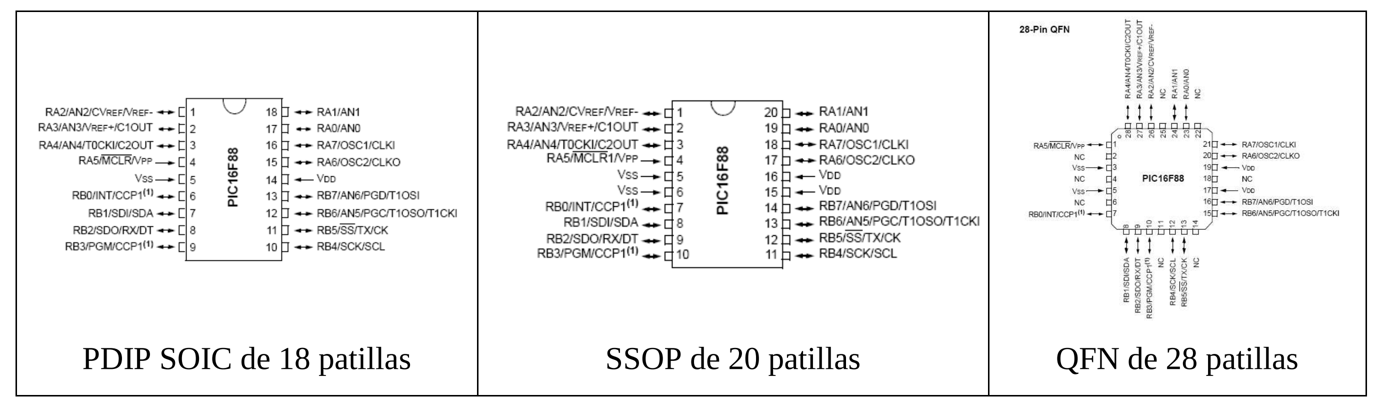


Figura 2: Encapsulados del PIC16F88

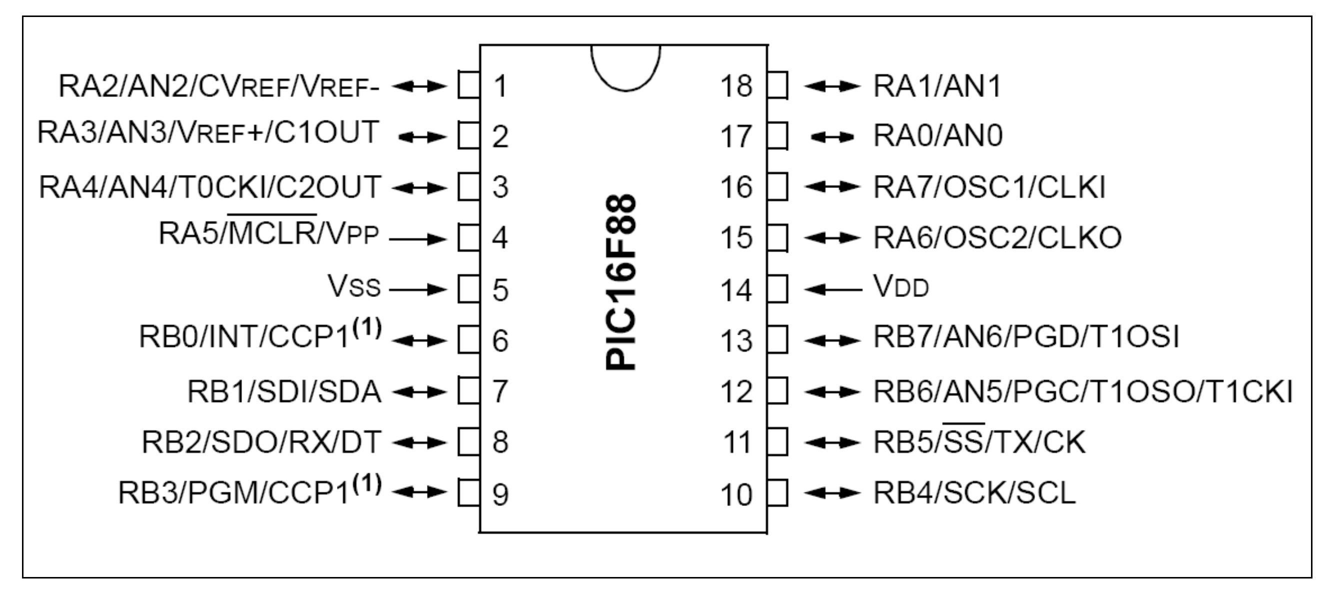


Figura 3: Patillaje del encapsulado PDIP

Nombre	**PIN**	**Tipo**	**Descripción**
RA0	17	I/O TTL	Pin TTL bidireccional
AN0		I ANALOG	Entrada analógica canal 0
RA1	18	I/O TTL	Pin TTL bidireccional
AN1		I ANALOG	Entrada analógica canal 1
RA2	1	I/O TTL	Pin TTL bidireccional
AN2		I ANALOG	Entrada analógica canal 2
CV_{REF}		O	Generador voltaje de referencia
V_{REF}-		I ANALOG	Entrada de tensión de referencia V^- del ADC
RA3	2	I/O TTL	Pin TTL bidireccional
AN3		I ANALOG	Entrada analógica canal 3
V_{REF}+		I ANALOG	Entrada de tensión de referencia V^+ del ADC
C1OUT		O	Salida del comparador 1
RA4	3	I/O ST	Pin Schmitt-Trigger bidireccional
AN4		I ANALOG	Entrada analógica canal 4
T0CKI		I ST	Entrada de impulsos del temporizador TMR0
C2OUT		O	Salida del comparador 2

RA5 MCLR’ V_{PP}	4	I ST I ST P	Pin de entrada Schmitt-Trigger Reset activo a nivel bajo. Entrada de tensión de programación
RA6 OSC2 CLKO	15	I/O ST O O	Pin Schmitt-Trigger bidireccional En modo Crystal Oscillator: Conexión al cristal de cuarzo En modo RC: ¼ de la frecuencia de reloj en OSC1 (ciclo de instrucción)
RA7 OSC1 CLKI	16	I/O ST I I	Pin Schmitt-Trigger bidireccional En modo Crystal Oscillator: Conexión al cristal de cuarzo En modo RC: Entrada de señal de reloj
RB0 INT CCP1	6	I/O TTL I ST I/O ST	Pin TTL bidireccional Interrupción externa Entrada para captura. salida para comparación y PWM
RB1 SDI SDA	7	I/O TTL I ST I/O ST	Pin TTL bidireccional Entrada de datos SPI™ E/S de datos I^2C™
RB2 SDO RX DT	8	I/O TTL O ST I I/O	Pin TTL bidireccional Salida de datos SPI™ Recepción asíncrona AUSART Detección síncrona AUSART
RB3 PGM CCP1	9	I/O TTL I/O ST I ST	Pin TTL bidireccional Habilitación de programación ICSP™ a baja tensión Entrada para captura. salida para comparación y PWM
RB4 SCK SCL	10	I/O TTL I/O ST I ST	Pin TTL bidireccional. Interrupción en cambio de estado. Entrada/salida de reloj para comunicación serie síncrona SPI™ Entrada de reloj para serie comunicación serie síncrona I^2C™
RB5 SS’ TX CK	11	I/O TTL I TTL O I/O	Pin TTL bidireccional. Interrupción en cambio de estado Selección de esclavo para SPI™ en modo esclavo Transmisión en AUSART modo asíncrono Reloj AUSART modo síncrono
RB6 AN5 PGC T1OSO T1CKI	12	I/O TTL I ANALOG I/O ST O ST I ST	Pin TTL bidireccional. Interrupción en cambio de estado. Entrada analógica canal 5 Reloj en programación ICSP™ y depuración en circuito Salida del oscilador del Timer1 Entrada de reloj externa para el Timer1
RB7 AN6 PGD T1OSI	12	I/O TTL I ANALOG I ST I ST	Pin TTL bidireccional. Interrupción en cambio de estado. Entrada analógica canal 6 Entrada de datos en programación ICSP™ y depuración en circuito Entrada del oscilador del Timer1
V_{SS}	5	P	Masa
V_{DD}	14	P	Alimentación

Tabla 2: Descripción de los pines de PIC16F88

3. ESTRUCTURA INTERNA Y ORGANIZACIÓN DE LA MEMORIA

3.1. DIAGRAMA DE BLOQUES

Como se ha mencionado antes, este dispositivo usa las arquitecturas RISC y Hardvard. Esto significa que:

1. El hardware es menos complejo que para los dispositivos de arquitectura CISC.
2. Memoria de instrucciones y de datos están separadas y usan buses distintos.

En la *figura 4* se muestra el diagrama de bloques del PIC 16F88.

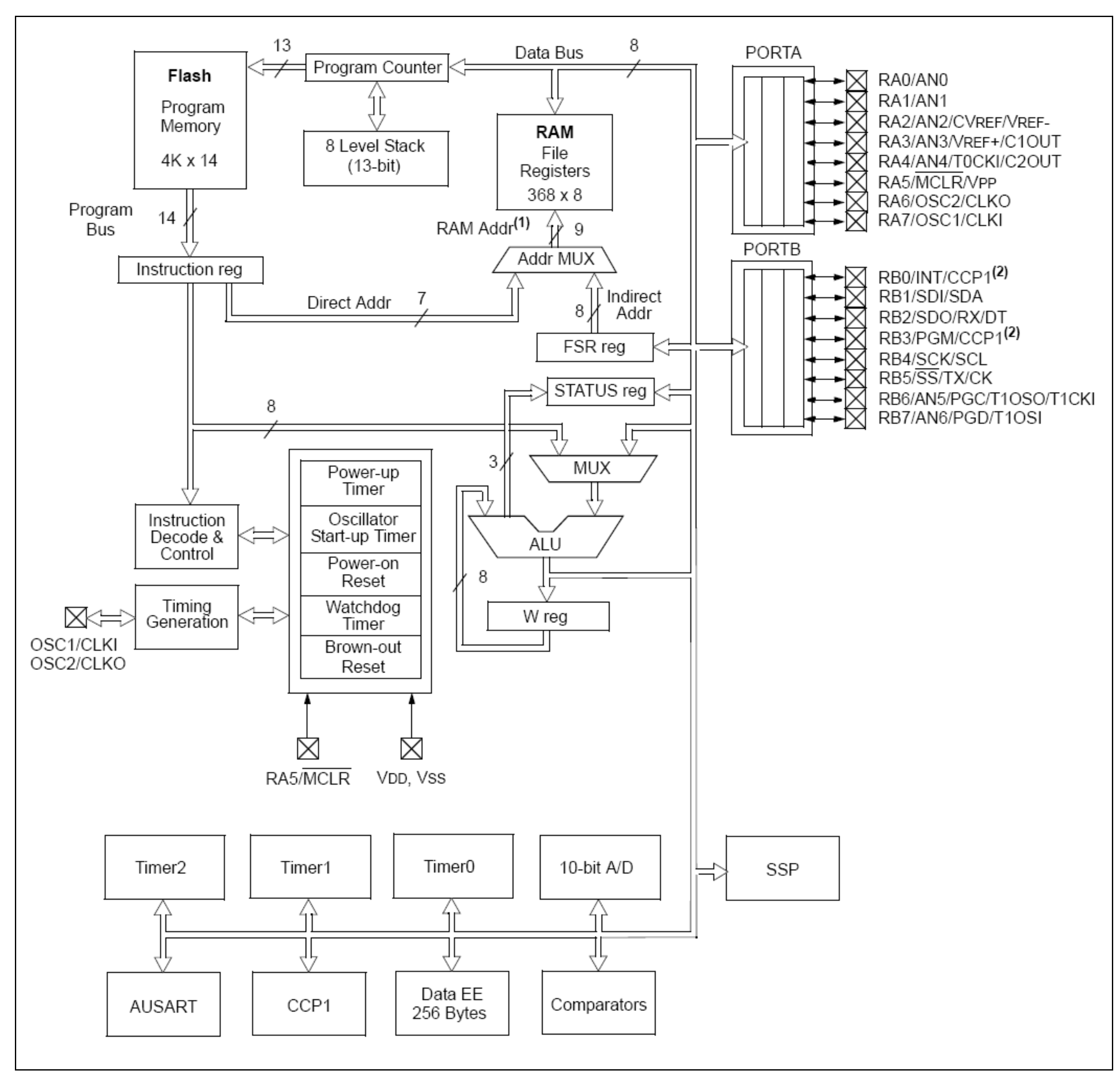


Figura 4: Diagrama de bloques del PIC 16F88

3.2. MEMORIA DE INSTRUCCIONES

En la parte superior izquierda del diagrama podemos apreciar la memoria de instrucciones de 4K. Ojo, esta memoria no tiene 4KB, ya que un byte son 8 bits y las palabras de esta memoria ocupan 14 bits. Lo que tiene es 4096 palabras de 14 bits.

También podemos observar que su bus de direcciones es de 13 bits, lo que significa que el número total de instrucciones que puede direccionar es de 2^{13} = 8K, aunque solo hay implementados 4K. Acceder a una de las direcciones superiores (MSB = 1), supone acceder a la misma dirección con el MSB = 0. Ejemplo: las direcciones 0C3Fh y 1C3Fh hacen referencia a la misma posición física de memoria.

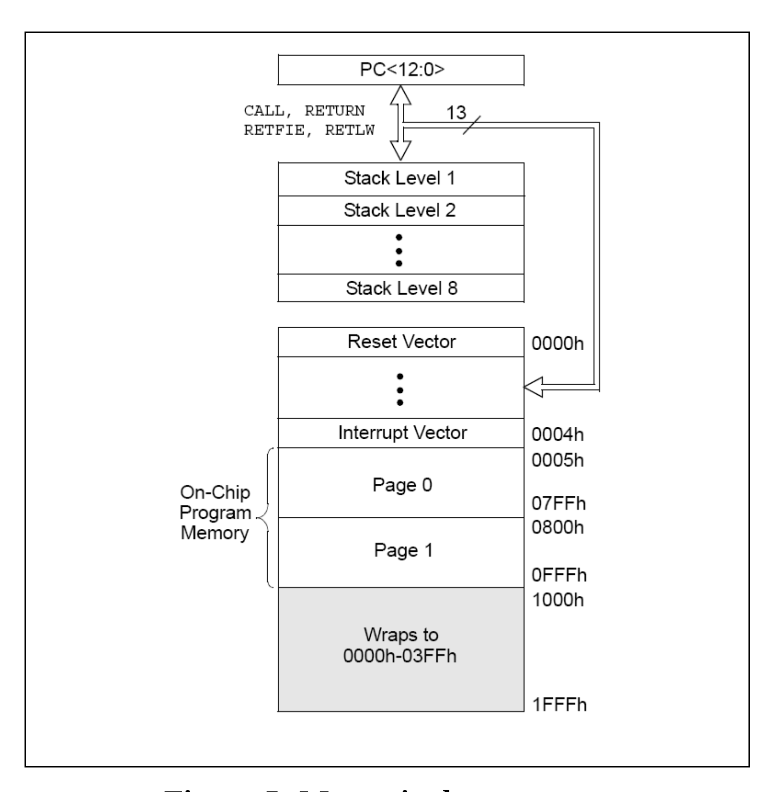


Figura 5: Memoria de programa

En la *figura 5* podemos ver la forma en que está organizada la memoria de instrucciones. El vector de reset se halla en la posición 0000h. En esta posición está la primera instrucción que leerá el micro en el momento de arrancar. El vector de interrupción está en la dirección 0004h. Si usamos interrupciones en nuestro programa tendremos que tener cuidado con esto, ya que en el momento de producirse una interrupción, el PC saltará automáticamente a esta dirección, por lo que nuestro programa tendrá que tener en esta dirección la rutina de interrupciones o su dirección de salto. Además, el programa principal, que debe comenzar en la posición 0000h, pasa por la dirección 0004h y esto deberá ser tenido en cuenta. Todo esto se explicará con mayor detalle en el capítulo dedicado a las interrupciones.

El resto de posiciones de memoria está organizado en dos páginas (0005h–07FFh y 0800h–0FFF). Al acceder a las posiciones a partir de 1000h, se accede a las anteriores.

Este dispositivo dispone de una pila hardware de 8 niveles para almacenar la dirección de retorno en caso de salto a subrutinas o de interrupción. La pila es manejada por la unidad de control y es inaccesible al programador tanto para leer como para escribir. Es difícil que se dé un desbordamiento de la pila al tener ésta 8 niveles, pero si se da, no hay ningún flag que lo indique, por lo que no debemos preocuparnos por esto, ya que no podemos hacer nada para evitarlo.

3.3. MEMORIA DE DATOS

En la parte derecha podemos ver el bus de la memoria de datos. Este bus si es de 8 bits. Los registros están mapeados en la memoria de datos, lo que quiere decir que cada uno tiene asignada una dirección de memoria y se accede a ellos como a una posición de memoria cualquiera. También los periféricos están conectados directamente al bus de datos, como se puede observar en la parte inferior de la imagen, y también se accede a ellos mediante registros mapeados en la memoria, tanto para E/S de datos como para configuración. Al bus de datos están conectadas las patillas que forman los puertos A y B y que son la vía de comunicación del integrado con el mundo exterior.

Entre memoria y registros tenemos un total de 512 posiciones, lo que hace necesario un bus de direcciones de 9 bits. En realidad, en las instrucciones solo se codifican 7 bits, debido a que no se podría codificar un número mínimo de instrucciones necesarias en 14 bits si se tienen que dedicar 9 de ellos al direccionamiento.

Entonces ¿Cómo se accede a todas las direcciones de memoria con solo 7 bits? La memoria está dividida en 4 bancos de 128 posiciones de memoria/registros, por lo que solo se necesitan 7 bits para acceder a cada una de las posiciones de una página dada. Para seleccionar uno de los cuatro bancos, uno de los registros (STATUS) que veremos más adelante, tiene dos bits RP0 y RP1. La unidad de control obtendrá la dirección completa a partir de los 7 bits de la instrucción y el contenido de estos 2 bits. Gracias a esto, la dirección solo ocupa 7 bits y se puede incluir sin ocupar demasiado en una instrucción de 14 bits. Como contrapartida, el programador debe mediante software cambiar los dos bits RP0 y RP1 cada vez que tenga que cambiar de banco de memoria a la hora de leer o escribir datos. En la *tabla 3* se puede ver la configuración de estos bits para cada selección de banco.

RP1:RP0	banco
00	0
01	1
10	2
11	3

Tabla 3: Selección de bancos de registros

3.4. MAPA DE MEMORIA

En los cuatro bancos de memoria se encuentran mapeados los registros específicos, los puertos de E/S y la memoria de uso general. *En la figura 6* se puede apreciar el mapa de memoria del PIC16F88.

Hay una serie de direcciones que merecen una dedicación especial. Las direcciones que aparecen en gris no están implementadas, por lo que no tendrá ningún efecto escribir en ellas y se leerán siempre como 0. La primera posición de memoria de cada banco no corresponde a ningún registro físico, si no que se usa para el direccionamiento indirecto que se explicará más adelante. En el banco 3 tenemos dos direcciones de memoria reservadas que no deben ser modificadas.

El resto de posiciones de memoria corresponden a registros específicos o a registros de propósito general. La memoria total disponible para datos es de 96 bytes en cada banco excepto en el 1 que es de 80 bytes, lo que hace un total de **368 bytes**. Las últimas 16 direcciones del banco 0 (70h – 7Fh) están mapeadas en el resto de bancos, lo que nos puede resultar útil para colocar ahí aquellas variables que se usen con frecuencia y evitar tener que cambiar de banco por software cada vez que queramos acceder a uno de esos datos. Lo mismo sucede con determinados registros a los que se accede con frecuencia como STATUS, FSR, INTCON… esto redunda en un código más compacto y legible al no tener que realizar un cambio de banco cada vez que queremos acceder a uno de estos registros.

	File Address		File Address		File Address		File Address
Indirect addr.(*)	00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
	07h		87h		107h		187h
	08h		88h		108h		188h
	09h		89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved(1)	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved(1)	18Fh
T1CON	10h	OSCTUNE	90h	General Purpose Register 16 Bytes	110h	General Purpose Register 16 Bytes	190h
TMR2	11h		91h				
T2CON	12h	PR2	92h				
SSPBUF	13h	SSPADD	93h				
SSPCON	14h	SSPSTAT	94h				
CCPR1L	15h		95h				
CCPR1H	16h		96h				
CCP1CON	17h		97h				
RCSTA	18h	TXSTA	98h				
TXREG	19h	SPBRG	99h				
RCREG	1Ah		9Ah				
	1Bh	ANSEL	9Bh				
	1Ch	CMCON	9Ch				
	1Dh	CVRCON	9Dh				
ADRESH	1Eh	ADRESL	9Eh				
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h
			EFh		16Fh		1EFh
		accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h-7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

Figura 6: Mapa de memoria

3.5. VALORES INICIALES

Varios registros del microcontrolador, se utilizan para configurar el funcionamiento general del mismo o los periféricos que vamos a utilizar. En temas posteriores se verá esto con mayor profundidad. De momento, solo diremos que es importante que configuremos correctamente dichos registros escribiendo el valor correspondiente.

No obstante, en el momento de arrancar el PIC, todos los registros se inician a un valor por defecto. Es conveniente conocer dicho valor, ya que puede suceder que utilicemos líneas de código para configurar una determinada característica, y que dichas lineas sean totalmente inncesarias debido a que el microcontrolador ya ha iniciado los registros con dicho valor a la hora de arrancar.

En las hojas de características del fabricante se puede ver dicho valor de inicio. A lo largo de esta guía iremos diciendo, cuando sea relevante, el valor de inicio del registro correspondiente.

4. JUEGO DE INSTRUCCIONES

El PIC16F88 tiene un juego de instrucciones reducido, pero bastante ortogonal. Se dice que un juego de instrucciones es ortogonal cuando cualquier instrucción puede utilizar cualquier elemento de la arquitectura como fuente o destino. Todas las instrucciones ocupan 14 bits con el operando implícito en el código de instrucción.

Todas las instrucciones duran un ciclo máquina correspondiente a cuatro ciclos de reloj, a excepción de las de salto condicional que pueden durar dos si se cumple la condición y de las de salto incondicional que duran siempre dos ciclos. Esto facilita el cálculo de tiempos para aplicaciones en tiempo real, ya que para una frecuencia de oscilación típica de 4MHz ($t = 0.25\mu s$), cada instrucción durará $4 \cdot 0.25\mu s = 1\mu s$ y cada instrucción de salto $2\mu s$, por lo que calcular el tiempo real que tarda en ejecutarse el código es bastante sencillo.

Podemos clasificar las instrucciones disponibles en tres categorías básicas:

- Operaciones orientadas a byte.
- Operaciones orientadas a bit.
- Operaciones con literales y de control.

Como vimos en el diagrama de bloques, disponemos de un registro de trabajo W. Dicho registro se va a usar siempre como uno de los operandos de la ALU. Los resultados de una operación se pueden almacenar bien en la posición de memoria del operando, o bien en W. Además todas las constantes o literales que usemos, se deben cargar en este registro para luego transferirse a cualquier otro sitio.

Los operandos pueden ser de dos tipos fundamentales: Literales o direcciones de memoria. Los literales solo se pueden usar como operandos para el registro W, o bien como direcciones de salto para las instrucciones de salto. Los literales con los que trabaja W son de 8 bits. En cuanto a las direcciones de memoria, como se dijo en el apartado 3.3., van a ser de 7 bits (recordar que hay 4 bancos de 128 posiciones) y se pueden especificar bien mediante la dirección, bien mediante una etiqueta, o bien mediante uno de los mnemónicos de los registros. Por ejemplo: Al poner PORTA nos referimos a la dirección 05h del banco 0 (ver figura 6). Una advertencia: PORTA no es más que un mnemónico de 05h. Esto quiere decir que al escribir PORTA accedemos a la dirección 05h del banco en que estemos. Es tarea nuestra asegurarnos de que estamos en el banco 0, si no, al poner PORTA estaremos accediendo a la dirección 05h del banco en que estemos, que no tiene por que ser el 0.

A continuación se citan todas las instrucciones del dispositivo. Se usa la nomenclatura del compilador MPASM™. Los mnemónicos de los operandos se incluyen en la *tabla 4*.

Mnemónico	Significado
f	Dirección del registro (7 bits). Se puede poner también el mnemónico del registro en lugar de la dirección (PORTB, ANSEL…)
W	Registro de trabajo o acumulador. Siempre aparece en operaciones con dos operandos
b	Para operaciones orientadas a bit: ordinal del bit sobre el que se ejecuta la operación (0-7)
k	Constante, literal o etiqueta
d	Destino. Cuando es 0 el resultado se almacenará en el registro W, cuando es 1 el resultado se almacena en el registro especificado

Tabla 4: Significado de los mnemónicos

4.1. INSTRUCCIONES ARITMÉTICAS

Instrucción	Descripción
`ADDLW k`	Suma a W el literal de 8 bits k
`ADDWF f,d`	Suma W con el registro f
`DECF f,d`	Decrementa el contenido del registro f
`INCF f,d`	Incrementa el contenido del registro f
`SUBLW k`	Resta al literal de 8 bits k el contenido de W
`SUBWF f,d`	Resta al registro f el contenido de W

Tabla 5: Instrucciones aritméticas

Ejemplos:

`ADDWF PORTB, 1` Suma el contenido de PORTB y de W. El resultado se almacena en PORTB.

`INCF 50h, 0` Incrementa el contenido de la dirección 50h. El resultado se almacena en W.

`SUBLW 40` Resta al número decimal 40 el contenido de W. El resultado se almacena en W.

4.2. INSTRUCCIONES LÓGICAS

Instrucción	Descripción
`ANDLW k`	Hace un AND bit a bit entre W y el literal k
`ANDWF f,d`	Hace un AND bit a bit entre W y el registro f
`COMF f,d`	Complementa f (niega todos los bits)
`IORLW k`	Hace un OR bit a bit entre W y el literal k
`IORWF f,d`	Hace un OR bit a bit entre W y el registro f
`XORLW k`	Hace un XOR bit a bit entre W y el literal k
`XORWF f,d`	Hace un XOR bit a bit entre W y el registro f

Tabla 6: Instrucciones lógicas

Ejemplos:

`ANDLW 10101010b` Hace el producto lógico entre el contenido de W y el literal 10101010. El resultado se almacena en W.

`XORWF 6Ah, 0` Hace XOR bit a bit entre el contenido de la dirección 6Ah y W. El resultado se almacena en W.

4.3. INSTRUCCIONES ORIENTADAS A BIT

Instrucción	Descripción
`BCF f, b`	Pone a 0 el bit en la posición b del registro f
`BSF f, b`	Pone a 1 el bit en la posición b del registro f
`RLF f, d`	Rota f a la izquierda a través del bit de carry
`RRF f, d`	Rota f a la derecha a través del bit de carry
`SWAPF f,d`	Intercambia los bits 7..4 con los bits 3..0 de f

Tabla 7: Instrucciones orientadas a bit

Ejemplos:

`BCF TRISA, 5` El bit 5 del registro TRISA pasa a ser 0.

`RLF 55h, 0` Si en 55h hay 00111100 y en el bit de carry CB = 1, el resultado es 01111001 y CB = 0. El resultado se almacena en W.

`SWAPF PORTB, 1` Si PORTB = 11110000, el resultado es 00001111. El resultado se almacena en PORTB.

4.4. INSTRUCCIONES DE CARGA Y MOVIMIENTO DE DATOS

Instrucción	Descripción
`CLRF f`	Borra el contenido del registro f
`CLRW`	Borra W
`MOVF f,d`	Copia el contenido del registro f a W
`MOVWF f`	Copia el contenido de W al registro f
`MOVLW k`	Carga el literal de 8 bits k en W

Tabla 8: Instrucciones de carga y movimiento de datos

Ejemplos:

`MOVF PORTA, 0` Copia el contenido de PORTA a W. También se puede escribir MOVFW.

`MOVLW 34` Copia en W el número decimal 34.

4.5. INSTRUCCIONES DE CONTROL DE FLUJO (SALTO CONDICIONAL)

Instrucción	Descripción
`BTFSC f,b`	Salta la siguiente instrucción si el bit en la posición b del registro f es 0
`BTFSS f,b`	Salta la siguiente instrucción si el bit en la posición b del registro f es 1
`DECFSZ f,d`	Decrementa el contenido del registro f. Si el resultado es 0, salta la siguiente instrucción
`INCFSZ f,d`	Incrementa el contenido del registro f. Si el resultado es 0, salta la siguiente instrucción

Tabla 9: Instrucciones de salto condicional

Ejemplos:

```
BTFSC 66h, 3
MOVWF PORTA
INCF PORTA, 1
```

Comprueba el bit 3 de la posición 66h. Sí es 0, saltará la instrucción que va a continuación (MOVWF PORTA) y ejecutará la siguiente (INCF PORTA, 1).

`INCFSZ 70h, 0` Incrementa el contenido de la posición 70h y el resultado se almacena en W. Si es 0, se salta la siguiente instrucción y se ejecuta la posterior.

4.6. INSTRUCCIONES DE SALTO

Instrucción	Descripción
`CALL k`	Llamada a subrutina en la dirección k
`GOTO k`	Ir a la dirección de programa k
`RETFIE`	Regreso desde rutina de interrupción. Igual que RETURN, pero además hace GIE = 1
`RETLW k`	Retorno de subrutina devolviendo el literal de 8 bits k en W
`RETURN`	Retorno de subrutina

Tabla 10: Instrucciones de salto

Sobre las instrucciones de salto, decir que la dirección de salto va implícita en la instrucción y es de 11 bits, lo que permite un total de 2K. Recordemos que la memoria de programa estaba segmentada en dos bloques de 2K. Tenemos instrucciones de salto e instrucciones de retorno. Los saltos se pueden hacer con GOTO o con CALL. La diferencia es que CALL almacena la dirección de retorno en la pila (para poder regresar con una instrucción de retorno) y GOTO no.

Ejemplos:

`CALL producto` Desvía el flujo del programa a la dirección de la etiqueta producto. La dirección de la instrucción siguiente se almacena en la pila.

`GOTO bucle` Salta a la dirección de la etiqueta bucle. No almacena dirección de retorno.

`RETLW FFh` Regresa de la subrutina y guarda el número FFh en W.

4.7. OTRAS INSTRUCCIONES

Instrucción	Descripción
`CLRWDT`	Reinicia el contador WatchDog Timer
`NOP`	No operación
`SLEEP`	Lleva el micro a modo sleep (reloj detenido)

Tabla 11: Otras instrucciones

Ejemplos:

`NOP` No hace nada. Estas instrucciones se usan para temporización.

4.8. RESUMEN DE INSTRUCCIONES

En la *tabla 12* podemos ver el juego de instrucciones completo.

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	**f, d**	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	**f, d**	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	**f**	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	**-**	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	**f, d**	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	**f, d**	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	**f, d**	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	**f, d**	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	**f, d**	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	**f, d**	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	**f, d**	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	**f**	Move W to f	1	00	0000	lfff	ffff		
NOP	**-**	No Operation	1	00	0000	0xx0	0000		
RLF	**f, d**	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	**f, d**	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	**f, d**	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	**f, d**	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	**f, d**	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	**f, b**	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	**f, b**	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	**f, b**	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	**f, b**	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	**k**	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	**k**	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	**k**	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	**-**	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	**k**	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	**k**	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	**k**	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	**-**	Return from interrupt	2	00	0000	0000	1001		
RETLW	**k**	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	**-**	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	**-**	Go into Standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	**k**	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	**k**	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Tabla 12: Juego de instrucciones del PIC 16F88

5. REGISTROS ESPECIALES

Como se ha comentado, el PIC 16F88 dispone de zonas de memoria diferentes para instrucciones y datos. En la memoria de datos tenemos registros de propósito general (para almacenar datos) y también se encuentran mapeados los registros especiales del PIC (de configuración general, interrupciones, configuración de periféricos, de E/S…). Varios de estos registros tienen una función asociada con un determinado dispositivo o periférico. Así, tenemos registros para configurar y comunicarnos con los puertos, para los conversores A/D, para el bus I2C, para acceder a la EEPROM, para los temporizadores… A medida que vayamos viendo los distintos periféricos y características del PIC 16F88, iremos explicando los registros asociados a cada operación y como utilizarlos.

En este apartado nos vamos a centrar en los registros del PIC que no están asociados a ningún periférico específico, si no que están más relacionados con el funcionamiento general del dispositivo, o que se van a usar para configurar ciertas opciones. No se pretende hacer una descripción exhaustiva de todas las características del microcontrolador, si no que nos centraremos solo en aquellas absolutamente necesarias para comenzar a realizar aplicaciones. Para conocer el funcionamiento general más a fondo, recurrir a las hojas de características del fabricante. En la *figura 6* se puede apreciar la dirección de memoria en la que está mapeado cada registro.

5.1. STATUS

7	6	5	4	3	2	1	0
IRP	**RP1**	**RP0**	**TO'**	**PD'**	**Z**	**DC**	**C**

El registro STATUS o de estado está mapeado en todos los bancos de memoria en la dirección 03h, contiene los bits de selección de banco el estado del reset, y los flags de la ALU.

IRP: Selección de banco para direccionamiento indirecto. 0 – bancos 0 y 1, 1 – bancos 2 y 3. Se verá con más detalle al ver el direccionamiento indirecto (registros INDF y FSR). Este bit toma el valor 0 al inicio.

RP1-0: Selección de banco de registros para direccionamiento directo.

RP1:RP0	Banco
00	0
01	1
10	2
11	3

Tabla 13: Selección de bancos de registros

Estos dos bits de configuración merecen una mención especial. El cambio de banco es algo que tendremos que hacer con bastante frecuencia en nuestros programas. La forma de hacerlo es mediante estos bits. En el momento de arrancar el sistema, estos bits toman el valor 0 (bank0). A continuación se pone un ejemplo de cómo cambiar de banco.

```
BSF     STATUS, RP0    ; ponemos el bit RP0 de STATUS a 1
BCF     STATUS, RP1    ; y el bit RP1 a 0 (o sea, banco 1)
MOVLW   0x00           ; ponemos en TRISB todo ceros
MOVWF   TRISB          ; (pasando por W)
BCF     STATUS, RP0    ; para seleccionar banco 0 hacemos RP0 = 0
MOVLW   0x55           ; ahora escribimos en el registro PORTB
MOVWF   PORTB
```

Desde ensamblador se puede hacer la selección de banco de una forma más fácil con la instrucción `BANKSEL` que selecciona el banco de un registro dado, por ejemplo:

```
BANKSEL     TRISB          ; es lo mismo que:
BSF         STATUS, RP0    ; ya que TRISB está en el banco 1
BCF         STATUS, RP1    ; y así se selecciona directamente
```

TO': Motivo del TIME-OUT: 1 – después de encendido, instrucción WDT o SLEEP. 0 – desbordamiento del WDT.

PD': POWER DOWN: 1 – después de encendido o por instrucción WDT. 0 – Ejecución de la instrucción SLEEP.

Z: Bit de 0: Se pone a 1 si el resultado de una operación aritmético- lógica es 0.

DC: Se pone a 1 sí al ejecutar una operación de suma o resta, se produce un CARRY o BORROW del bit 3 al bit 4.

C: Se pone a 1 sí al ejecutar una operación de suma, se desborda el resultado. Se pone a 0 si al realizar una operación de resta el resultado es negativo.

5.2. OPTION_REG

7	6	5	4	3	2	1	0
RBPU'	**INTEDG**	**T0CS**	**T0SE**	**PSA**	**PS2**	**PS1**	**PS0**

Este registro está mapeado en los bancos 1 y 3 de memoria en la dirección 01h. Contiene varios bits de control para configurar varias opciones relacionadas con el PORTB, interrupciones, y TMR0.

RBPU': Este bit se usa para configurar el PULL-UP interno del PORTB. Con 1 el PULL-UP se desactiva, con 0 se puede activar independientemente para cada patilla. Esto se verá más a fondo cuando nos centremos en el estudio del PORTB. Por defecto está desactivado

INTEDG: Configuramos si la interrupción en el cambio en la patilla RB0/INT se produce en un flanco ascendente (1) o descendente (0). Se verá en el tema de interrupciones.

T0CS: Fuente de cambio en TMR0: 1- cambio de estado en la patilla RA4/T0CKI/C2OUT. 0- Contador interno. Se verá más a fondo en el tema de temporizadores.

T0SE: Con T0CS = 1, cuenta en flanco ascendente (0) o descendente (1) en la patilla RA4/T0CKI/C2OUT. Se verá más a fondo en el tema de temporizadores.

PSA: Asignación de la preescala (bits 210 de este mismo registro) al TMR0 (0) o al WatchDog Timer (1). Se verá más a fondo en el tema de temporizadores.

PS2-1-0: Rango de la preescala asignada al TMR0 o al WatchDog Timer. Se verá más a fondo en el tema de temporizadores.

Valor	TMR0	WDT
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Tabla 14: Preescalas para TMR0 y WDT

5.3. PCL Y PCLATH

Como se ha dicho, el contador del programa es de 13 bits (8K). Sin embargo los registros internos del microcontrolador son de 8 bits, por lo que se van a necesitar dos registros para almacenar la dirección de memoria. Estos dos registros son PCL y PCLATH, que se encuentran mapeados en todos los bancos en las posiciones 02h y Ah respectivamente. El registro PCL contiene los 8 bits inferiores de la dirección de la memoria de instrucciones. Los 5 bits restantes son los 5 bits de menos peso del registro PCLATH. Como se puede apreciar en la *figura 7*.

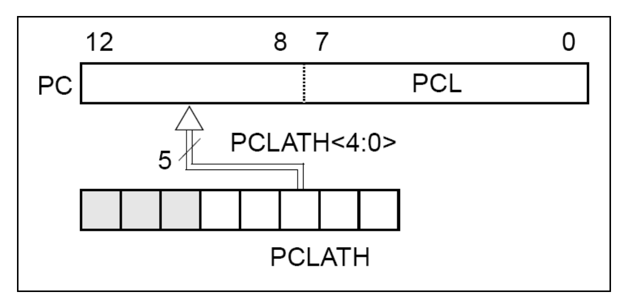


Figura 7: Dirección del PC

La forma en que se obtiene la dirección absoluta merece una explicación adicional. Al realizar un salto con GOTO o CALL, la instrucción incluye el código de operación y 11 bits de dirección… Pero al ser el PC de 13 bits ¿de dónde se obtienen los dos bits restantes?

Como se dijo en el apartado 3.2., la memoria de instrucciones se encuentra dividida en dos páginas de 2K (11 bits) cada una. La selección de una posición en una página se hace mediante los 11 bits de menos peso. Con los otros 2 bits seleccionamos la página en cuestión (en realidad solo con uno, ya que aunque el PIC16F88 puede direccionar hasta 8K, solo están implementadas físicamente 4K). Al usar un GOTO o CALL si el destino es una página distinta a la actual, debemos modificar los dos bits de más peso de la dirección (bits 3 y 4 de PCLATH) manualmente., ya que si no lo hacemos, el salto se producirá a una dirección de la página actual. Al volver de una subrutina o interrupción, esto no será necesario, ya que la pila es de 13 bits y ha almacenado la dirección de retorno completa. Pero al llamar, deberemos hacerlo siempre. No insistiremos más en esto, ya que los programas que realicemos usarán solo una página, pero es conveniente saberlo por si alguna vez necesitamos realizar programas que sobrepasen las 2K de tamaño.

Por último, añadir que se puede escribir en el registro PCL como en un registro cualquiera, por lo que podíamos implementar un GOTO escribiendo directamente en el PC los 8 bits de dirección (sabiendo muy bien lo que se hace, por supuesto).

5.4. INDF Y FSR

Los registros INDF y FSR se usan para direccionamiento indirecto. INDF no es un registro físico. Al leer o escribir en INDF, lo que hacemos es acceder a la dirección de memoria contenida en FSR. Por ejemplo: al poner en FSR 48h, al escribir en INDF lo que haremos será escribir en la dirección 48h. El direccionamiento indirecto es muy útil a la hora de trabajar con matrices o grupos de datos. El siguiente ejemplo, extraído de las hojas de características del fabricante, muestra como utilizar el direccionamiento indirecto para borrar las direcciones de memoria situadas entre 20h y30h:

```
MOVLW   0x20        ; Cargamos el registro FSR
MOVWF   FSR         ; con la dirección 20h
BORRAR
CLRF    INDF        ; borramos INDF (al principio la dirección 20h)
INCF    FSR, 1      ; incrementamos FSR (al principio pasa a 21h)
BTFSS   FSR, 4      ; si el bit 4 de FSR es 1 (dir 30h) saltamos
                    ; la siguiente instrucción (o sea, terminamos)
GOTO    BORRAR      ; borraremos la siguiente dirección 21h…
:                   ; aquí continúa el programa
:
```

Con FSR (8 bits) podemos direccionar 256 posiciones. Si recordamos, la memoria de datos está paginada en 4 bancos de 128 posiciones (9 bits en total). Esto significa, que para poder acceder a todas las posiciones necesitamos un bit más. Este bit es el bit 7 (IRP) del registro STATUS. Cuando IRP = 0 accedemos a los bancos 0 y 1; cuando sea 1 a los bancos 2 y 3. Recordar que cada banco tiene 128 direcciones, por lo que con FSR accedemos a direcciones de los dos bancos. Por ejemplo: Si IRP = 0, accederemos a los bancos 0 y 1. Al poner 20h accedemos a la posición 20h del banco 0. Para acceder a la posición 20h del banco 1, pondremos el bit de más peso de FSR a 1, o sea A0h.

5.5. OSCCON

7	6	5	4	3	2	1	0
	IRCFC2	**IRCFC1**	**IRCFC0**	**OSTS**	**IOFS**	**SCS1**	**SCS0**

El registro OSCCON se usa para configurar varios parámetros con el modo de operación del reloj del sistema. Se encuentra mapeado en la dirección 0E del banco 1. Una de las cosas que tendremos que configurar en este registro será la frecuencia de oscilación del oscilador interno cuando queramos usarlo. Recordemos que este dispositivo dispone de su propio oscilador interno, que evita tener que usar un cristal externo y permite dedicar esas patillas a otra función. El oscilador interno tiene una frecuencia interna regulable de hasta 8MHz. En el momento de arrancar el sistema, la frecuencia del oscilador interno es de 31.25KHz, aunque se puede modificar en tiempo de ejecución cambiando los bits correspondientes.

IRCFC2-1-0: Frecuencia del oscilador interno

Valor	F
000	31.25KHz
001	125KHz
010	250KHz
011	500KHz
100	1MHz
101	2MHz
110	4MHz
111	8MHz

Tabla 15: Frecuencias del oscilador interno

OSTS: Bit de estado del oscilador al arrancar el sistema. Con 1 el sistema está usando el reloj principal, con 0, de T1OSC o del reloj interno cuándo éste está configurado como secundario.

IOFS: Indica si la frecuencia del oscilador interno es estable (1) o inestable (0).

SCS1-0: Oscilador que se usará como reloj principal en modo los modos de ahorro de energía. Se puede usar el oscilador principal, definido en la palabra de configuración (CONFIG1) en los bits FOSC2 : FOSC0, la entrada TMR1, o el oscilador interno.

Valor	Oscilador
00	principal
01	T1OSC
10	osc interno
11	reserved

Tabla 16: Oscilador usado

A continuación se muestra el código necesario para configurar el oscilador interno para que funcione a una frecuencia de 4MHz:

```
MOVLW   b'01100000'    ; Tenemos que meter en OSCCON
MOVWF   OSCCON               ; el valor x110XX00
```

5.6. OSCTUNE

7	6	5	4	3	2	1	0
		TUN5	**TUN4**	**TUN3**	**TUN2**	**TUN1**	**TUN0**

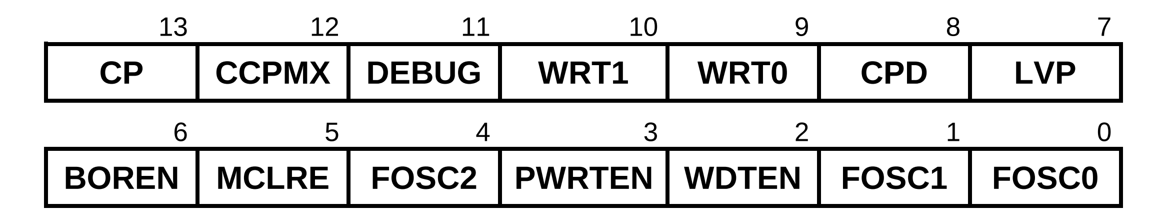
El oscilador interno del que dispone el PIC16F88, viene calibrado de fábrica. No obstante, esta frecuencia puede ser ajustada en un rango de ±12.5%

TUN5-0: Bits de ajuste de frecuencia.

Valor	F
011111	Frecuencia máxima (+12.5%)
011110	
..	
000001	
000000	frecuencia central (0%)
111111	
..	
100000	Frecuencia mínima (-12.5%)

Tabla 17: Ajuste de frecuencia

5.7. CONFIG1



Este registro merece una atención especial. Como se ve, al contrario que los anteriores, es un registro de 14 bits que se halla mapeado en la memoria de programa en la dirección 2007h. Nótese que esta dirección excede la dirección de memoria de programa máxima que se puede direccional con el PC (13 bits – 1111111111111 – 1FFFh), por lo que en tiempo de ejecución no podemos acceder a la

misma. Este registro se usa para programar distintas configuraciones para el dispositivo. Solo se puede escribir en tiempo de grabación, escribiendo lo que se llama la *palabra de configuración*, que definirá el comportamiento que tendrá nuestro dispositivo en diversos aspectos que veremos a continuación.

Estos bits **pueden programarse (se ponen a 0) o dejarse sin programar, en cuyo caso se quedan a 1**. Veremos que para activar unas propiedades tienen que estar a 1 y para otras a 0.

CP: Protección del código. Con 0 el código está protegido.

CCPMX: Selección de la patilla por la que se accede al módulo CCP1. Con 1 CCP1 está en la patilla RB0. Con 0 en la patilla RB3.

DEBUG: Al ponerse a 0, se permite la depuración en circuito (In-Circuit Debugger). En este modo las patillas RB6 y RB7 no se pueden usar para propósito general, ya que se necesitan para este fin.

WRT1-0: Protección de escritura en la memoria de programa.

Valor	protección
00	0000h - 0FFFh
01	0000h - 07FFh
10	0000h - 00FFh
11	desactivada

Tabla 18: protección contra escritura

CPD: Protección de la memoria de datos. Con 0 está protegida.

LVP: Habilitación de programación a bajo tensión (Low-Voltage Programming). Al activarse con un 1, la patilla RB3/PGM se usa para LVP.

BOREN: Habilitación de Brown-out Reset. Esto consiste en reiniciar el dispositivo cuando la tensión de alimentación desciende de un determinado nivel durante un determinado tiempo. Cuando esto sucede, el sistema no se reinicia hasta que la alimentación vuelve a alcanzar unos valores óptimos de nuevo. Esta propiedad se puede activar poniendo este bit a 1.

MCLRE: Selección de la función del pin RA5/MCLR/VPP. Con 1 esta patilla se usará para reset y no podrá ser usada para otra cosa. Con 0 la entrada de reset MCLR se une internamente a Vcc y la patilla puede ser usada para cualquiera de sus funciones.

PWRTEN: Habilitación del Power-up Timer. Es un temporizador que añade un delay fijo de 72ms en el arranque del sistema. Esto es útil para conseguir que las tensiones de alimentación y el oscilador alcancen su punto óptimo de funcionamiento y así evitar que haya un comportamiento erróneo del programa debido a una baja tensión de alimentación al principio, por ejemplo. Con esto podemos evitar tener que usar un reset para iniciar correctamente el sistema en la mayoría de los casos, lo que permite liberar la patilla MCLR para otros fines. Poniendo este bit a 0 activamos esta propiedad.

WDTEN: Habilitación del perro guardián (Watchdog Timer). No vamos a ver en este curso el perro guardián. Solo decir, que consiste en un temporizador interno que si se desborda, provoca una interrupción. Debe ser reiniciado por el programa continuamente para evitar esto. Su utilidad consiste en evitar que el programa entre en bucles infinitos por cualquier motivo. Si esto sucede, no se actualiza el WDT y se desborda. Es crítico elegir los puntos de programa donde debemos actualizar el WDT. Se puede habilitar poniendo este bit a 1.

FOSC2-0: Selección del modo de oscilador. Nosotros usaremos generalmente el oscilador interno. Las características y conexionado de los demás modos se pueden consultar en las hojas de características del fabricante.

Valor	Modo	Características
000	LP	Bajo consumo
001	XT	Cristal de cuarzo
010	HS	Cristal de alta velocidad
011	ECIO	Externo con E/S en RA6
100	INTIO2	Oscilador interno
101	INTIO1	Oscilador interno con salida fosc/4 en RA6
110	RCIO	Circuito RC externo
111	RC	Circuito RC externo con fosc/4 en RA6

Tabla 19: Modos de oscilador

A continuación vamos a ver que palabra de configuración debemos programar para un determinado comportamiento del sistema. Lo que vamos a hacer es lo siguiente: Vamos a usar el oscilador interno sin salidas, El módulo CCP1 en la patilla RB0, La función de reset desactivada y de las funciones del microcontrolador habilitaremos el Power-Up timer y deshabilitaremos la protección de código, la depuración en circuito, la protección contra escritura en la memoria de programa, la protección de la memoria de datos, La función LVP, EL Brown-out reset y el Watchdog timer. En la *tabla 20* se puede observar la forma de conseguir esto. Podemos observar que la palabra de configuración sería: 11 1111 0001 0000 = 3F10h. Esta será la configuración que más usemos.

BIT	Valor	Característica
13	1	Sin protección de código
12	1	CCP en patilla RB0
11	1	Sin depuración en circuito
10	1	Sin protección contra escritura en mem de programa
9	1	Sin protección contra escritura en mem de programa
8	1	Sin protección de memoria de datos
7	0	Sin LVP
6	0	Sin Brown-Out Reset
5	0	Sin patilla de reset
4	1	Oscilador interno
3	0	Power-Up timer activado
2	0	Watchdog Timer desactivado
1	0	Oscilador interno
0	0	Oscilador interno

Tabla 20: Palabra de configuración

6. PUERTOS A Y B

Además de todos los periféricos de mayor o menor complejidad que posee el PIC16F88, orientados para realizar un gran número de tareas, muchas de ellas muy específicas, tenemos los puertos de E/S PORTA y PORTB que serán los que más usemos. La configuración de los puertos se hará con los registros TRISA, TRISB y ANSEL.

6.1. PORTA Y PORTB

PORTA y PORTB son dos puertos de E/S digital de 8 bits cada uno, cuyas patillas se pueden configurar como salidas o entradas independientemente (excepto RA5 que solo se puede usar como entrada). PORTA y PORTB se hallan mapeados en las direcciones 05h y 06h respectivamente del banco 0 de memoria (PORTB está mapeado además en la misma dirección del banco 2). Para acceder a ellos basta con poner su nombre. Al acceder a estas posiciones de memoria, debemos recordar que no es una posición de memoria normal, si no un puerto de E/S, por lo que las operaciones de lectura y escritura van a tener ciertas particularidades. Por ejemplo, si el puerto está configurado como entrada, al realizar una operación de escritura en el puerto, ésta no tendrá efecto. Si por el contrario, está configurado como salida, al leer lo que leeremos será el último valor que hayamos escrito.

Nos referiremos a las patillas de manera independiente como RA0, RA1…RA7 y RB0, RB1,…RB7. Cada una de estas patillas se corresponde con un bit del registro, RA0/RB0 con el de menos peso y RA7/RB7 con el de más, de tal forma que si en PORTA tenemos F0h, las patillas RA7…RA4 tendrán un 1 y las patillas RA3…RA0 tendrán un 0.

6.2. ANSEL

Hemos vistos que el PIC 16F88 tiene varios periféricos multiplexados en sus 18 patillas. Cada uno de estos periféricos puede ser de entrada o salida, analógico o digital. Es evidente que la forma en que se tratan las señales analógicas es muy distinta a como se hace con las digitales, por lo que el dispositivo lleva un hardware de entrada/salida distinto para cada tipo de señal analógica o digital.

Las entradas pueden ser analógicas o digitales. La selección del tipo de entrada se hace mediante el registro ANSEL mapeado en la dirección 1Bh del banco 1. En este registro se hallan los bits de control de todas las entradas analógicas del dispositivo, de tal forma que se pueden activar

independientemente poniendo un 1 o desactivar con un 0, por lo que en este último caso podemos usar las patillas como E/S digital.

7	6	5	4	3	2	1	0
	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0

Cada uno de los bits del registro ANSEL conecta la entrada analógica correspondiente, que como se puede observar en el patillaje del integrado, AN0…AN4 se corresponden con RA0…RA4, AN5 con RB6 y AN6 con RB7. Es importante **reseñar que estos bits están a 1 por defecto**, por lo que si queremos usar alguna de estas patillas como E/S digital, tendremos que poner a 0 los bits correspondientes antes de hacerlo. A continuación se ve un ejemplo de cómo configurar todas las patillas como E/S digital (o sea, desactivar las entradas analógicas):

```
BSF     STATUS, RP0     ; ponemos el bit RP0 de STATUS a 1
BCF     STATUS, RP1     ; y el bit RP1 a 0 (o sea, banco 1)
MOVLW   0x00            ; ponemos en ANSEL todo ceros
MOVWF   ANSEL           ; (pasando por W)
```

6.3. TRISA Y TRISB

Puesto que los puertos A y B son bidireccionales, debe existir alguna forma de indicar al dispositivo si queremos usarlos como entrada o como salida. Existen dos registros asociados con PORTA y PORTB que permiten configurarlos. Los registros TRISA y TRISB se hallan mapeados en las direcciones 05h y 06h respectivamente del banco 1 de memoria, al igual que PORTA y PORTB en el banco 0 (TRISB también está mapeado en la misma posición del banco 3). Cada uno de los bits de estos registros se corresponde con una patilla, de tal forma que **al poner en esa posición un 1, esa patilla será de entrada y al poner un 0 será salida**. De inicio están **Configurados para entrada** (11111111), o sea, que deberemos poner a 0 los bits correspondientes a aquellas patillas que queramos utilizar como salida antes de utilizarlos. Para configurar las patillas 0, 1, 2, 6 y 7 del puerto B como entradas y 3, 4 y 5 como salidas pondríamos:

```
BSF     STATUS, RP0     ; ponemos el bit RP0 de STATUS a 1
BCF     STATUS, RP1     ; y el bit RP1 a 0 (o sea, banco 1)
MOVLW   b’11000111’     ; ponemos en TRISB 11000111
MOVWF   TRISB           ; (pasando por W)
```

Recordar que la patilla 5 del puerto A es solo entrada, por lo que cualquier escritura en el bit 5 de TRISA no tendrá efecto, ya que este bit estará siempre a 1.

A continuación se muestra un programa completo que trabaja con puertos. Este programa lee las patillas del puerto A y lo manda al puerto B:

```
BANKSEL     ANSEL          ; seleccionamos el banco 1
MOVLW       0x00           ; ponemos en ANSEL todo ceros
MOVWF       ANSEL          ; (E/S digital)
MOVWF       TRISB          ; ponemos TRISB a 0 (todo salidas)
MOVLW       0xFF           ; ponemos TRISA todo unos
MOVWF       TRISA          ; o sea, PORTA es entrada
BANKSEL     PORTA          ; seleccionamos el banco 0
MOVFW       PORTA          ; copiamos PORTA a W
MOVWF       PORTB          ; y W a PORTB
```

7. TEMPORIZADORES

7.1. TEMPORIZADORES Y CONTADORES

Para cualquier aplicación en tiempo real, se hace necesaria una temporización. Por temporizador entendemos un periférico que a partir del reloj del microprocesador puede contar un determinado intervalo de tiempo. En un dispositivo como un microcontrolador, orientado al control, poseer uno o más temporizadores es algo totalmente imprescindible (piénsese en cualquier sistema que necesite usar un reloj, un semáforo, un regulador de velocidad…).

Además de la temporización, hay muchas aplicaciones que necesitan contar impulsos externos. Un contador es un dispositivo que cuenta los impulsos que apliquemos en una determinada entrada, algo que también se va a usar en muchas aplicaciones (Tacómetro, pulsímetro…) por lo que los microcontroladores deberán incluir también contadores.

Hay aplicaciones que van a necesitar de ambos periféricos. Por ejemplo: Piénsese en utilizar un PIC como frecuencímetro. Puesto que la frecuencia es el número de ciclos por unidad de tiempo, para poder medir esta magnitud necesitaremos contar los impulsos que se producen en una entrada (contador) en un determinado intervalo de tiempo (temporizador). El PIC 16F88 dispone de tres temporizadores que se pueden configurar para realizar varias funciones y que veremos a continuación detalladamente.

7.2. TIMER0

El Timer0 es un temporizador/contador de 8 bits. Se halla mapeado en la dirección 01h (**bank0**) y se identifica por el mnemónico **TMR0**. Dispone además de preescala programable, flanco programable y puede producir una interrupción al desbordarse (FFh – 00h) configurándolo para que lo haga en el registro INTCON (se verá en el tema de interrupciones). Podemos contar cualquier número entre 00h y FFh, ya que su registro asociado es de lectura-escritura (para contar hasta 100, por ejemplo, cargamos el valor 155 y al pasar de 255 a 0 habrá contado hasta 100). El modo de operación del Timer0 se controla mediante el registro **OPTION_REG**. Recordemos el contenido de este registro. Los bits asociados al TMR0 son 5-0.

7	6	5	4	3	2	1	0
RBPU’	**INTEDG**	**T0CS**	**T0SE**	**PSA**	**PS2**	**PS1**	**PS0**

T0CS: Configuración como contador o temporizador: 1- contador de impulsos en la patilla RA4/T0CKI/C2OUT. 0- Contador de ciclos de instrucción (temporizador).

T0SE: En modo contador: cuenta en flanco ascendente (0) o descendente (1).

PSA: Asignar la preescala (bits 210 de este mismo registro) al TMR0 (0) o al WDT (1).

PS2-1-0: Rango de la preescala asignada al TMR0.

Valor	preescala
000	1 : 2
001	1 : 4
010	1 : 8
011	1 : 16
100	1 : 32
101	1 : 64
110	1 : 128
111	1 : 256

Tabla 21: Preescalas para TMR0

En modo temporizador, el Timer0 se incrementa en cada ciclo de instrucción a no ser que utilicemos preescala, en cuyo caso se incrementará cada N ciclos de instrucción, donde N es el valor de la preescala.

Al realizar una escritura, el contador se inhibe durante dos ciclos, lo que deberá ser tenido en cuenta a la hora de realizar temporizaciones en tiempo real. Debemos ser muy cuidadosos con la precisión a la hora de realizar aplicaciones en tiempo real, ya que aunque no lo hayamos pensado, los relojes son muy precisos. Por ejemplo: un error del 0.1%, que en cualquier magnitud que no fuese tiempo sería despreciable, en una aplicación en tiempo real va a suponer un desvío de 3.6 segundos a la hora 86.4 segundos al día… algo totalmente inadmisible. Tendremos que jugar con la cuenta y la preescala para que el tiempo que dura la cuenta sea lo más aproximado posible al que queremos contar. Lo veremos en el tema de interrupciones.

Como contador, usamos la patilla 3 que se corresponde con RA4/AN4/T0CKI/C2OUT. Se pueden contar flancos de subida o de bajada en función del bit T0SE. El siguiente fragmento de código cuenta los impulsos en la patilla T0CKI:

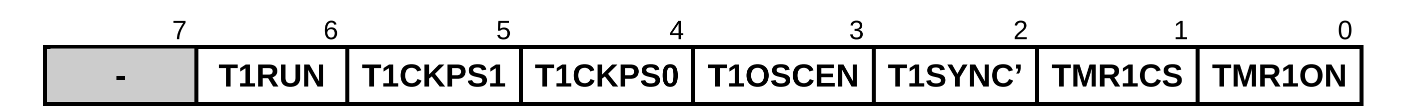
```
BANKSEL     OPTION_REG
MOVLW       b'00100010'    ; TMR0 contador de flancos ascendentes
MOVWF       OPTION_REG     ; en T0CKI con preescala 1:8
CLRF        TRISB          ; ponemos TRISB a 0 (todo salidas)
BANKSEL     PORTB          ; seleccionamos el banco 0
CLRF        TMR0
bucle
MOVFW       TMR0           ; copiamos TMR0 a W
CLRF        TMR0           ; reiniciamos cuenta
MOVWF       PORTB          ; y copiamos W a PORTB
GOTO        bucle          ; repetimos para siempre
END
```

7.3. TIMER1

El PIC 16F88 dispone de un temporizador de 16 bits, el Timer1, mapeado en las posiciones 0Eh y 0Fh del **bank0** (**TMR1L** y **TMR1H** respectivamente) Cuenta de 0000h a FFFFh y puede producir una interrupción al desbordarse (FFFFh – 0000h) configurándolo para que lo haga poniendo a 1 el bit TMR1IE del registro PIE1 (dirección 8C, 0C del bank1). El modo de operación del Timer1 se controla mediante el registro **TMR1CON** (10h bank0). Podemos configurarlo bien como temporizador, contador síncrono y contador asíncrono. En modo contador se incrementa en cada flanco ascendente en la patilla RB6/AN5/PGC/T1OSO/**T1CKI**. En modo temporizador, TMR1 se incrementa con cada ciclo de instrucción. El TMR1 se utiliza también para el módulo CCP que se estudiará mas adelante. El TMR1 también se puede usar como entrada secundaria de reloj en modo bajo consumo conectando un cristal de 32KHz en las patillas correspondientes.

A continuación se ve como configurar el TMR1 con el registro TMR1CON.

TMR1CON



7	6	5	4	3	2	1	0
-	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC’	TMR1CS	TMR1ON

T1RUN: Bit de estado del reloj del sistema. 1- Reloj en la entrada del TMR1. 0- Reloj derivado de otra fuente.

T1CKPS1-0: Preescala asignada a la entrada T1CKI (en modo contador).

Valor	preescala
00	1 : 1
01	1 : 2
10	1 : 4
11	1 : 8

Tabla 21: Preescalas para TMR1

T1OSCEN: Habilitación del oscilador en TMR1. 1- Activado, 0- Desactivado

T1SYNC': Sincronización de reloj externa del TMR1. 1- No sincronizar entrada externa de reloj, 0- Sincronizar. Este bit solo tiene efecto cuando TMR1CS = 1.

TMR1CS: Fuente del TMR1. 1- Reloj externo en la patilla RB6/AN5/PGC/T1OSO/**T1CKI**. 0 – Reloj Interno (F_{OSC}/4).

TMR1ON: Detener/Activar la cuenta del TMR1. 1- Activar, 0- Detener

Puesto que el TMR1 es de 16 bits, debemos tomar ciertas precauciones a la hora de escribir o leer los registros afectados, ya que se puede dar un paso de cuenta de uno a otro mientras leemos o escribimos. Por ejemplo, si a la hora de leer un dato el contenido de los registros es 12 FF, si leemos en primer lugar, por ejemplo el registro de menos peso, leeremos FF, pero puede darse el caso de que al hacer la siguente lectura, se halla incrementado la cuenta a 13 00, por lo que al leer el registro de mas peso, obtendremos 13. El valor final que tenemos es 13 FF, que es incorrecto. A continuación se pone la forma correcta de realizar una operación de lectura o escritura para evitar esto.

Escribir en TMR1:

```
CLRF        TMR1L     ; Borrar TMR1L (Así no se modificará TMR1H)
MOVLW       HI_BYTE   ; Cargar el valor en el byte de mas peso
MOVWF       TMR1H
MOVLW       LO_BYTE   ; Cargar el byte de menos peso
MOVWF       TMR1H     ; Write Low byte
```

Leer de TMR1:

```
MOVFW       TMR1H     ; Leer byte alto
MOVWF       TMPH      ; Escribir en registro temporal
MOVFW       TMR1L     ; Leer byte bajo
```

```
MOVWF       TMPL      ; Escribir en registro temporal
MOVFW       TMR1H     ; Leer byte alto de nuevo
SUBWF       TMPH, W   ; Restar la segunda lectura de la primera
BTFSC       STATUS, Z ; Si es cero (Z=1) no ha habido salto. OK
GOTO        continuar ; Lectura correcta. Continuar código
; Si no son iguales, ha habido acarreo de TMR1L a TMR1H.
; Leemos de nuevo (ahora será correcto)
MOVFW       TMR1H     ; Leer byte alto
MOVWF       TMPH      ; Escribir en registro temporal
MOVFW       TMR1L     ; Leer byte bajo
MOVWF       TMPL      ; Escribir en registro temporal
continuar             ; Continuar la ejecución
  …         …
```

El ejemplo siguiente implementa un contador de segundos usando como base de tiempos el TMR1. Se usa el oscilador interno a 250KHz, la frecuencia de instrucción será 62.5KHz, por lo que cuando el TMR1 cuente hasta 62500, habrá pasado un segundo. Lo mejor es contar desde 3037 (0BDDh) para así detectar el 0 en el temporizador, que será más sencillo.

```
segundos    equ  0x20          ;variable para almacenar los seg.

ORG 0
  bsf       STATUS,RP0         ;seleccionar bank1
  bsf       OSCCON,IRCF1       ;oscilador a 250KHz (IRCF(2:0)=010
  bcf       STATUS,RP0         ;seleccionar bank0
  bsf       T1CON,TMR1ON       ;activar tmr1
bucle
  movf      TMR1H              ;comprobar si ha pasado un segundo
  btfss     STATUS,Z           ;(TIMER1 se desborda ->TMR1H = 0)
  goto      bucle              ;si no lo es, continuar
  movlw     0x0B               ;cargar inicio de cuenta (0BDDh)
  movwf     TMR1H
  movlw     0xDD
  addwf     TMR1L
  incf segundos
  goto      bucle
  END
```

7.4. TIMER2

El Timer2 es un temporizador de 8bits, con preescala y postescala programables. Se encuentra mapeado en la posición 11h del bank0 (**TMR2**). Cuenta de 00h a FFh. Este registro es el que se usará como base de tiempos para la modulación PWM del módulo CCP1, como veremos en el tema dedicado a dicho módulo. El registro **TMR2** es de lectura y escritura, y se pone a 00h cuando se produce un reset en cualquier dispositivo. El reloj de entrada es el ciclo máquina del PIC ($4 \cdot T_{ck}$) y se le puede asignar una preescala variable de 1:1 a 1:16. El Timer2 también tiene una postescala programable entre 1:1 y 1:16. Es importante tener en cuenta que los valores de pre y postescala son reiniciados cada vez que escribimos en **TMR2** o en su registro de control **T2CON**.

El Timer2 tiene un registro de periodo **PR2**, mapeado en la dirección 12h del bank1, de tal forma que en el momento en que **TMR2** iguala al contenido de **PR2**, **TMR2** se reinicia. Esto es útil a la hora de ajustar el periodo de la onda PWM, como veremos más adelante.

Cuando el **TMR2** se desborda (teniendo en cuenta la postescala) puede producir una interrupción, configurándolo para que lo haga poniendo a 1 el bit 1 (**TMR2IE**) del registro **PIE1** (dirección 8C, 0C del bank1). Cuando se produce una desbordamiento, se pone a 1 el bit 1 del registro **PIR1** (**TMR2IF**).

El modo de operación del Timer2 se controla mediante el registro **T2CON** (12h bank0). A continuación se ve como configurar el **TMR2** con el registro **T2CON**.

T2CON

7	6	5	4	3	2	1	0
-	**TOUTPS3**	**TOUTPS2**	**TOUTPS1**	**TOUTPS0**	**TMR2ON**	**T2CKPS1**	**T2CKPS0**

TOUTPS3-0: Postscala asignada a la salida del TMR2.

Valor	preescala
0000	1 : 1
0001	1 : 2
:	: :
:	: :
1111	1 :16

Tabla 22: Postscalas para TMR2

TMR2ON: Detener/Activar la cuenta del TMR1. 1- Activar, 0- Detener

T2CKPS1-0: Preescala asignada a la frecuencia de entrada ($f_{osc}/4$).

Valor	preescala
00	1 : 1
01	1 : 4
1X	1 : 16

Tabla 22: Preescalas para TMR2

8. MÓDULO CCP

El PIC 16F88 incluye un módulo CCP (Capture, Compare, PWW). La entrada/salida de dicho módulo está en la patilla 6 (RB0/INT/**CCP1)** o en la 9 (RB3/PGM/**CCP1**). Podemos usar una u otra programando el bit 12 (**CCPMX**) del registro **CONFIG1**. 0 – RB0, 1 – RB3. Recordar que este registro es usado para la palabra de configuración, por lo que este valor se programa en tiempo de grabación y no puede ser cambiado. La patilla por defecto es **RB0**.

El módulo CCP1 tiene una resolución de 16 bits para captura y comparación, y de 10 bits para PWM. Los registros donde está dicho valor son **CCPR1L** y **CCPR1H**, que se encuentran mapeados en las posiciones de memoria 15h y 16h respectivamente del bank0. Puede realizar tres funciones distintas:

Captura: Se utiliza el Timer1. Cuando se produce un cambio en la patilla CCP1, se pasa el contenido del Timer1 al registro CCP1. Esta función puede ser útil para realizar, por ejemplo, un frecuencímetro.

Comparación: Cuando el Timer1 alcanza el valor de CCPR1, se produce un cambio en la patilla y una interrupción. Puede ser útil para hacer un generador de frecuencia.

PWM: Se obtiene una señal PWM (*Pulse Width Modulation*) en la patilla CCP1.

En esta guía solo veremos como utilizar este módulo para obtener una señal PWM. Para el resto de funcionalidades ir a las hojas de características del fabricante.

8.1. PRINCIPIOS DE LA MODULACIÓN PWM

Una de las aplicaciones de los microcontroladores es el control de la cantidad de energía que debe llegar a una determinada carga. El control más simple que tenemos es el control todo-nada: utilizamos un transistor conectado al elemento de control, como se muestra en la figura 8.1, de tal forma que al dar un 1 a la salida el transistor pasa a saturación y conduce. Con un 0 el transistor estará en corte. De esta forma, podemos encender o apagar el dispositivo a controlar.

Pero en ocasiones, el control todo-nada no es suficiente. Piénsese por ejemplo en un motor cuya velocidad de giro queremos controlar. Para hacer esto, debemos hacer que la tensión que llega al motor no sea constante si no variable. Una forma de conseguir esto es entregar una tensión variable en la base, lo que hará que podamos obtener una corriente variable por la carga. Pero el hecho de obtener una tensión variable de esta manera lleva acarreados una serie de inconvenientes:

1. El transistor va a trabajar en activa. Va a tener tensión y corriente a la vez, lo que hará que consuma potencia.
2. El circuito de control debe ser capaz de entregar una tensión variable en lugar de los dos valores 0 y 1 lógico, lo que va a hacer que necesitemos un circuito más complejo.

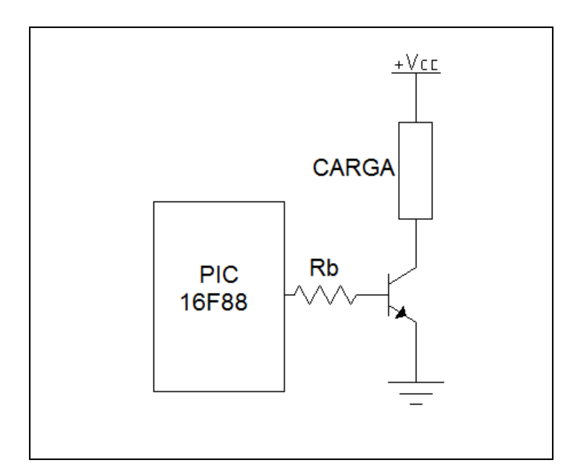


Figura 8.1: Circuito de potencia

Para resolver estos inconvenientes, existe el control PWM. Son las siglas de *Pulse Width Modulation*, o en castellano. *Modulación de la anchura del impulso*. La modulación PWM se basa en el control todo-nada, por lo que el control se puede realizar con una salida digital, y el transistor no consumirá potencia. La forma de conseguir una tensión variable de esta manera, es utilizar una señal cuadrada de una determinada frecuencia con el ciclo de trabajo variable, de tal forma que al enviar un 1 lógico, el transistor conducirá, y con el 0 no. El ciclo de trabajo (*duty cicle*) se define como la parte proporcional del periodo de la señal en la que ésta está a nivel alto: $CT = \frac{T_H}{T_H + T_L}$. Se puede poner en tanto por 100. El valor medio de la señal modulada en PWM será directamente proporcional al ciclo de trabajo: $V_{MED} = V_{CC} \cdot CT$, de tal forma que variando dicho valor, variaremos el valor medio de la señal. Esto se puede apreciar con mayor detalle en la figura 8.2.

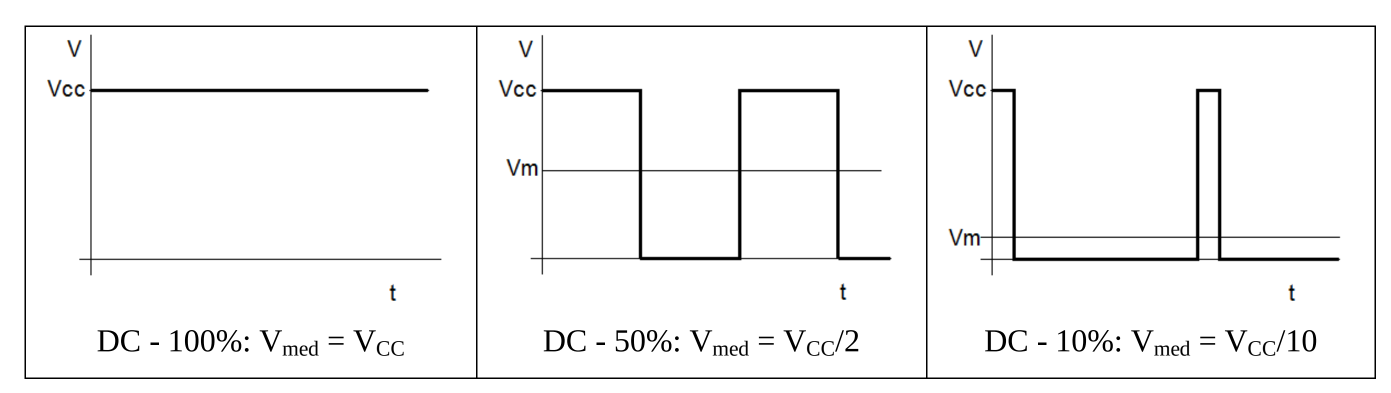


Figura 8.2: Distintas tensiones medias con PWM

Es por esto por lo que la modulación PWM es ampliamente usada a la hora de controlar cargas. El control PWM se puede implementar por software de una forma sencilla. Para generar una onda rectangular, basta con poner durante un tiempo la patilla a 1 y posteriormente a 0. En función de la relación entre ambos estados, tendremos un determinado ciclo de trabajo. No obstante, esto no será necesario ya que el PIC 16F88 lleva implementada un módulo específico para generar ondas moduladas en PWM.

8.2. CCP1CON

El control del módulo CCP1 del PIC se lleva a cabo mediante el registro CCP1CON. Este registro se encuentra mapeado en la dirección 17h del bank0. Y tiene la estructura que se muestra a continuación.

7	6	5	4	3	2	1	0
-	**-**	**CCP1X**	**CCP1Y**	**CCP1M3**	**CCP1M2**	**CCP1M1**	**CCP1M0**

CCP1X-Y: Los dos bits menos significativos de la modulación PWM. En los modos de captura y comparación no se usan

.

CCP1M3-0: Selección del modo del módulo CCP1. Se citan a continuación todos los modos posibles, aunque nosotros solo lo configuraremos para PWM.

Valor	modo
0000	Deshabilitar módulo
0100	Captura cada flanco de bajada
0101	Captura cada flanco de subida
0110	Captura cada 4 flancos de subida
0111	Captura cada 16 flancos de subida
1000	Comparación. CCP1 =1 cuando coincide
1001	Comparación. CCP1 = 0 cuando coincide
1010	Comparación. CCP1 sin cambios
1011	Comparación especial. Inicio del módulo A/D
11XX	PWM

Tabla 8-1: Configuración del modulo CCP1

8.3. MODULACIÓN PWM CON EL PIC16F88

Vamos a ver a continuación como se consigue la modulación PWM en el PIC 16F88. Como se ha dicho anteriormente, el módulo CCP1 es de 16 bits situados en los registros CCPR1H y CCPR1L. Para el modo PWM se usan 10 bits solamente. Mediante estos bits indicamos el ciclo de trabajo de la señal PWM: 0000000000 para el 0%, 1111111111 para el 100%. Estos 10 bits se ubican de la siguiente forma: Los 8 bits de mayor peso, los escribiremos en CCPR1L, que actuará como maestro mientras que CCPR1H actuará como esclavo, y los dos bits de menor peso se escribirán en los bits CCP1X-Y del registro CCP1CON. Podemos escribir en CCPR1L en cualquier momento, pero solo pasan a CCPR1H cada vez que empieza un ciclo. Si no necesitamos una resolución muy elevada, podemos prescindir de estos dos bits, pero teniendo en cuenta lo siguiente: Estos bits por defecto están a 0 por lo que para llegar al 100% de ciclo de trabajo, deberemos ponerlos a 1, ya que si prescindimos de ellos, el valor de la onda PWM será 1111111100 que es el 99.6%. De la misma forma, para conseguir un ciclo de trabajo del 0% posteriormente, deberán ponerse a 0.

El periférico asociado al modulador PWM es el TMR2. Si recordamos, es un contador de 8 bits con preescala y postescala. El periodo de la onda va a ser fijado por el TMR2, jugando con la preescala y con el registro de periodo PR2. Recordemos que el TMR2 se reinicia al alcanzar el valor del registro PR2. El periodo de la onda será:

$$T_{PWM} = (PR2+1) \cdot 4T_{OSC} \cdot \mathrm{Pr}Escala_{TMR2}$$

Es necesario hacer un par de aclaraciones sobre esto. En primer lugar, vemos que el modulador PWM es de 10 bits, mientras que el TMR2 es de solo 8 bits. Los dos bits de menos peso se obtienen de un reloj interno inaccesible al programador, de tal forma que al final el número total de bits usados son 10. Otro hecho a tener en cuenta, es que el TMR2 no cuenta siempre hasta 255, si no hasta que alcanza el valor en PR2. Esto quiere decir que a la hora de escribir en CCPR1L un número para obtener un determinado ciclo de trabajo, debemos tener en cuenta que el 100% no será el máximo (FF), si no el valor escrito en PR2. Si nos pasamos se producirá un overflow y el modulador no funcionará bien. Ejemplo: Si escribimos en PR2 el valor 200 (11001000), el valor a escribir en CCPR1L para un ciclo de trabajo del 100% no será 255 (11111111), si no 201 (11001000), y para un ciclo de trabajo del 50%, no deberá ser 128, si no 100 (01100100). Nótese que aunque jugar con el registro PR2 nos de una gran flexibilidad a la hora de fijar el periodo de la onda, podemos perder resolución, que puede ser crítico si ponemos un valor muy bajo.

El siguiente fragmento de código muestra como obtener una onda de 1KHz con un ciclo de trabajo del 50% a partir de una frecuencia de reloj de 1MHz:

```
bsf      STATUS,RP0     ;seleccionamos bank1
clrf ANSEL              ;E/S digital
clrf TRISB              ;PORTB   salida
bsf      OSCCON,IRCF2   ;f = 1MHz
movlw    .249           ;cargar en PR2 249
movwf    PR2            ;Tpwm = (249+1)·4·1us·1 = 1ms
bcf      STATUS,RP0     ;volver a bank0

bsf      T2CON,TMR2ON   ;poner en funcionamiento TMR2
bsf      CCP1CON,CCP1M3 ;configurar módulo CCP1 para PWM
bsf      CCP1CON,CCP1M2 ;poniendo a 1 los bits CCP1M3,2
movlw    .125      ;Para 100%: 249+1=250. Para el 50%: 125
movwf    CCPR1L    ;escribir valor en CCPR1L

goto     $         ;y repetir una y otra vez
```

9. CONVERSOR A/D

9.1. SEÑALES ANALÓGICAS Y DIGITALES

En la actualidad, prácticamente todos los procesos se realizan sobre señales digitales. Como se ha visto, el tratamiento y/o transmisión digital de señales presenta múltiples ventajas con respecto el tratamiento analógico: mayor inmunidad al ruido, posibilidad de encriptar/comprimir las señales, procesos definidos por software, almacenamiento… está claro que a la hora de trabajar con cualquier tipo de señal, la mejor opción es hacerlo de forma digital.

Pero para poder hacer esto, las señales deben de ser digitales, y esto no es algo que siempre ocurra. Por ejemplo, Cuando queremos medir cualquier tipo de magnitud (tensión, temperatura, presión, humedad…) en ocasiones el transductor usado nos dará la información de manera analógica. Por ejemplo, el sensor de temperatura LM35 ofrece a su salida una tensión de 10mV/ºC, de tal forma que para 25.4ºC ofrecerá aproximadamente 0.254V, para 25.5ºC dará 0.255V… queda claro que es una señal continua, o sea una magnitud variable con infinitos valores, en lugar de ser una sucesión de unos y ceros que tome únicamente unos valores determinados. No es una señal digital, por lo que a priori no puede ser tratada por un sistema microprogramable.

Como hemos visto, existen unos circuitos que permiten conectar ambos mundos. Tenemos conversores de analógico a digital (A/D o ADC) para expresar una señal analógica en forma digital y conversores de digital a analógico (D/A o DAC) para obtener magnitudes analógicas a partir de señales digitales. Está claro que si queremos tratar señales analógicas de forma digital es necesario disponer de estos circuitos.

El PIC 16F88 trae incorporado un conversor A/D de 7 canales con una resolución de 10bits, lo cual nos va a permitir trabajar con señales analógicas sin necesitar ningún tipo de circuitería externa. Tiene varias opciones configurables, como selección del canal, frecuencia de muestreo, tensión de referencia…

A continuación se analizan los registros relacionados con el conversor A/D.

9.2. ANSEL

7	6	5	4	3	2	1	0
	ANS6	**ANS5**	**ANS4**	**ANS3**	**ANS2**	**ANS1**	**ANS0**

Cuando queremos digitalizar una señal, nos referimos, evidentemente, a una señal analógica. Cuando queremos introducir una señal analógica en el PIC, hay que configurar dicha entrada como tal. Recordemos que esto se hacía en el registro ANSEL, mapeado en la dirección 1Bh del banco 1. Se activan las entradas analógicas poniendo a 1 el bit correspondiente. AN0…AN4 se corresponden con RA0…RA4, AN5 con RB6 y AN6 con RB7. Es **importante** reseñar que también deberemos configurar la patilla correspondiente como entrada en el registro correspondiente (TRISA o TRISB).

9.3. ADCON0

7	6	5	4	3	2	1	0
ADCS1	**ADCS0**	**CHS2**	**CHS1**	**CHS0**	**GO/DONE'**	**-**	**ADON**

El registro ADCON0 contiene bits de control del conversor A/D. Se halla mapeado en la dirección 1Fh del bank0 y en el se hallan los bits de control que se pueden modificar con más frecuencia.

ADCS1-0: Frecuencia de muestreo del conversor A/D. Podemos obtener la frecuencia del reloj del sistema dividida entre varios valores, o bien usar el propio oscilador interno del ADC. Al usar el reloj del sistema, obtenemos la frecuencia configurando además el bit ADCS2 del registro ADCON1. En la siguiente tabla se pueden ver todas las posibilidades.

Valor	ADCS2=0	ADCS2=1
00	$F_{osc}/2$	$F_{osc}/4$
01	$F_{osc}/8$	$F_{osc}/16$
10	$F_{osc}/32$	$F_{osc}/64$
11	F_{RC}	F_{RC}

Tabla 22: Frecuencias del ADC

Hay que reseñar, que este tiempo es por bit. La conversión completa llevará 10 veces mas tiempo. Para configurar la frecuencia debemos recordar el teorema de Nyquist:

$f_S > 2 \cdot f_{\max}$. También tenemos que tener en cuenta que el proceso de conversión lleva un cierto tiempo, por lo que no debemos muestrear demasiado rápido, ya que puede que no de tiempo a que se completen las conversiones. Lo ideal, según los datos del fabricante, es que el tiempo de conversión por bit esté entre 1.6µs y 6.4µs. El oscilador interno del ADC, tiene un tiempo de adquisición típico de 4µs, aunque presenta bastante deriva, pudiendo variar entre 2µs y 6µs. Si no es necesario tener un mucha precisión con la frecuencia de muestreo, el propio oscilador interno de del ADC es más que suficiente.

CHS2-0: Selección del canal analógico a procesar. Aunque el PIC 16F88 disponga de 7 entradas analógicas, solo tenemos un conversor A/D, por lo que debemos indicarle de cual de los 7 canales debe leer la entrada analógica. En la tabla siguiente se puede ver como realizar dicha selección.

Valor	CANAL	PATILLA
000	AN0	RA0
001	AN1	RA1
010	AN2	RA2
011	AN3	RA3
100	AN4	RA4
101	AN5	RB6
110	AN6	RB7

Tabla 23: Selección del canal

GO/DONE': Bit de estado de la conversión. Al poner este bit a 1, el módulo ADC comienza la conversión. Cuando la conversión haya terminado, este bit se pondrá automáticamente a 0, por lo que cumple a la vez las funciones de activación de conversión y flag.

ADON: Habilitación del módulo ADC. Debemos ponerlo a 1 cuando queramos usar el conversor A/D.

9.4. ADCON1

7	6	5	4	3	2	1	0
ADFM	**ADCS2**	**VCFG1**	**VCFG0**	**-**	**-**	**-**	**-**

Este registro se halla mapeado en la posición 1Fh del bank1 (9Fh). Se usa para configurar parámetros como formato del resultado o referencias de tensión.

ADFM: Formato del resultado de la conversión en ADRESH y ADRESL. Con 1 justificado a la derecha y con 0 a la izquierda. Se verá más a fondo en el apartado siguiente dedicado a estos registros.

ADCS2: Al ponerlo a 1, el reloj usado para conversión se divide entre 2. Se ha visto en el apartado anterior.

VCFG1-0: Configuración del voltaje de referencia. Podemos definir las tensiones de referencia V_{REF+} y V_{REF-} para el intervalo de conversión. Podemos usar bien las tensiones de alimentación V_{DD} y V_{SS}, o utilizar tensiones de referencia externas en las patillas RA2 (V_{REF-}) y/o RA3 (V_{REF+}). Si decidimos hacer esto, debemos configurar dichas patillas como entrada analógica. En la tabla siguiente se pueden ver las distintas posibilidades. Hay que tener en cuenta, que no podemos utilizar cualquier rango de valores. En las hojas de características el fabricante recomienda que entre la tensión mínima y la máxima haya más de 2V para un comportamiento correcto del módulo.

Valor	V_{REF+}	V_{REF-}
00	V_{DD}	V_{SS}
01	V_{DD}	V_{REF-}
10	V_{REF+}	V_{SS}
11	V_{REF+}	V_{REF-}

Tabla 24: Tensiones de referencia

9.5. ADRESL, ADRESH

Los registros ADRESH y ADRESL se encuentran mapeados en la posición de memoria 1Eh de los bancos 0 y 1 respectivamente (1Eh y 9Eh absolutas). En estos registros es donde se hallará el resultado de la conversión una vez finalizada. Se necesitan dos registros, ya que el conversor es de 10 bits y los registros de 8. Los bits de más peso estarán en ADRESH y los de menos peso en ADRESL. Entre los dos registros hay 16 bits, por lo que 6 de ellos no contendrán ninguna información. Podemos elegir dos maneras de justificar el resultado mediante el bit ADFM de ADCON1.

Con ADFM = 0 el resultado en ADRESH y ADRESL estará justificado a la izquierda. Esto quiere decir que los 10 bits de la conversión se organizarán de la siguiente forma: los ocho de más peso se situarán en ADRESH y los dos restantes en los dos bits de más peso de ADRESL. Los seis bits de menos peso de este registro serán 0.

Con ADFM = 1 el resultado en ADRESH y ADRESL estará justificado a la derecha. Esto quiere decir que los dos bits de más peso se situarán en los dos bits de menos peso de ADRESH y los ocho restantes en ADRESL. Los seis bits de más peso de ADRESH serán 0.

Cuando queramos usar una conversión de menor resolución (8 bits) pondremos ADFM = 0, de tal forma que los 8 bits de más peso estarán en ADRESH, y no necesitaremos mirar ADRESL que además está en el bank1. Por defecto ADFM = 0. Esta forma de proceder es bastante frecuente, ya que en muchas ocasiones 8 bits son más que suficientes.

9.6. PROCESO DE CONVERSIÓN

Para usar el conversor A/D del PIC 16F88 se deben realizar las siguientes acciones:

1. Habilitar la entrada analógica que queremos utilizar (en ANSELy TRISA-B).
2. Seleccionar dicha entrada como fuente para el DAC.
3. Configurar las tensiones de referencia.
4. Configurar reloj.
5. Si se van a utilizar, configurar interrupciones.
6. Configurar justificación del resultado.
7. Encender el módulo
8. Comenzar conversión.
9. Esperar a que termine.
10. Leer resultado.

Tenemos dos formas de saber cuando la conversión está completa. Bien comprobando el estado del bit GO/DONE’ de ADCON0 (cuando se ponga a 0 la conversión estará completa), o bien, si utilizamos interrupciones, esperar a que se produzca la interrupción y se salte automáticamente a la rutina de interrupciones. Esto se verá con más detalle en el tema de interrupciones.

Para comenzar una nueva conversión, se pone a 1 el bit GO/DONE’. Si nuestro programa va a realizar varias cosas aparte de la conversión, es conveniente apagar el módulo para así ahorrar energía

y activarlo solo cuando se necesite. Si lo hacemos así, es necesario esperar un tiempo entre que se activa el módulo y se inicia la conversión. ($2T_{AD}$ es suficiente).

A continuación se incluye un ejemplo de uso del conversor A/D.

```
inicio
   bsf          STATUS,RP0
   movlw  b'01100000'    ;frecuencia de reloj = 4MHZ
   movwf  OSCCON
   movlw  b'00000001'    ;activar la entrada analogica AN0 (RA0)
   movwf  ANSEL
   bsf    TRISA,0        ;RA0 entrada
   movlw  b'01000000'    ;configurar ADCON1:0-justificado izquierda
   movwf  ADCON1         ;0- No div por 2, 00-referencia Vdd y Vss.
   bcf    STATUS,RP0

   ;configurar ADCON0:   00:4MHz/2 (2MHz),  000- AN0, 00- nada,0- ON
   clrf   ADCON0

bucle
   bsf    ADCON0,0  ;activar modulo AD
   call   DELAY_20u ;retardo de 20us
   bsf    ADCON0,2  ;comenzar conversión
convert
   btfsc  ADCON0,2  ;esperar a que termine la conversión
   goto   convert   ;bit 2 de ADCON0 (GO/DONE’) =  0
   bcf    ADCON0,0  ;desactivar modulo

   …                ;resto del programa
   …

   goto   bucle

END
```

10. EEPROM

10.1. CARACTERÍSTICAS

En ocasiones, necesitamos almacenar datos en nuestros programas de forma no volátil. Tomemos el ejemplo de una cerradura electrónica. La clave de acceso, aunque pueda modificarse por software, debería ser la misma después de un corte de alimentación. Como ya se ha comentado, el PIC 16F88 dispone de una memoria EEPROM de 256 bytes para almacenar datos permanentemente. Hay que hacer una aclaración sobre esto. El PIC 16F88 dispone de una memoria de programa ROM FLASH de 4Kw de 14 bits. A la hora de almacenar datos, se puede utilizar indistintamente la EEPROM o la misma memoria de programa. De hecho, la forma de acceder a una u otra es prácticamente igual. La única diferencia consiste en especificar a cual de las dos queremos acceder. A la hora de utilizar variables no volátiles (o sea, escribir en memoria), es conveniente utilizar la EEPROM, debido a que soporta mayor número de ciclos de Lectura/escritura, tiempo de almacenamiento, además de una mayor sencillez a la hora de realizar operaciones de escritura, pero conviene saber que no tenemos por qué limitarnos a 256 bytes de memoria no volátil, tenemos muchísima memoria de programa para almacenar datos no volátiles (preferiblemente constantes escritas en tiempo de programación).

En este apartado vamos a dedicarnos solo al manejo de la EEPROM de datos. Para obtener información sobre como acceder a datos en la memoria de programa consultar las hojas de características del fabricante. Obsérvese que el hecho de poder modificar la memoria de programa desde el propio programa nos puede permitir hacer código automodificable, que se actualice para ser más óptimo, rápido… en resumen, programas que aprendan. Lo más avanzado que puede haber en programación.

Un factor importante a tener en cuenta, es que para poder escribir datos en ambas memorias es conveniente que el PIC se alimente a 5V. Tensiones inferiores podrían ser insuficientes a la hora de grabar datos. Los registros relacionados con el acceso a la EEPROM de datos y la memoria de programa son los mismos y se hallan en los bancos 2 y 3 de memoria. Tenemos registros de configuración/control, de direcciones y de datos. Todos ellos se pueden ver a continuación.

10.2. EEDATA Y EEADR. EEDATH Y EEADRH

Mediante estos registros accedemos a la EEPROM. EEDATA es el registro mediante el cual escribimos y leemos datos de la EEPROM. Aquí escribiremos los datos a almacenar y aquí aparecerán

los datos a la hora de leer. Se halla mapeado en la dirección 10Ch de bank2. Para especificar la posición de la EEPROM a la que deseamos acceder se escribe en el registro EEADR. Es un registro de solo escritura mapeado en la dirección 10Dh de bank2.

Como ya se ha comentado, estos registros se usan también para acceder a la memoria de programa. La diferencia es que la longitud de palabra son 14 bits en lugar de 8 y además tenemos 4096 posiciones, por lo que necesitaremos 12 bits para especificar la dirección en lugar de 8. Los bits de más peso necesarios para el acceso a la memoria de programa se hallarán en los registros EEDATH Y EEADRH. Que se hallan mapeados en las posiciones de bank2 10Eh y 10Fh respectivamente.

10.3. EECON1 Y EECON2

Estos registros se hallan mapeados en las direcciones 18Ch y 18Dh de bank3 respectivamente. EECON2 no es un registro físico. Se utiliza solo para la secuencia de escritura en la EEPROM. En EECON1 se hallan los bits de configuración de la EEPROM. Su contenido es el citado a continuación.

EECON1



7	6	5	4	3	2	1	0
EEPGD	-	-	FREE	WRERR	WREN	WR	RD

EEPGD: Selección de acceso a ROM de programa (1) o EEPROM de datos (0).

FREE: Solo para memoria de programa. En la siguiente operación de escritura escribir en una posición (0) o borrar 32 posiciones (1).

WRERR: Flag de error en escritura. Se pone a 1 si no se ha podido terminar el ciclo de escritura debido a un reset o WDT.

WREN: Habilitar la escritura en la EEPROM. Si se deja a 0 los accesos serán de solo lectura.

WR: Iniciar ciclo de escritura. Este bit solo se puede poner a 1. Se pone a 0 por hardware automáticamente en cuanto el ciclo de escritura termina.

RD: Iniciar ciclo de lectura. Al igual que el anterior, se pone a 1 por software y a 0 por hardware al finalizar el ciclo de lectura.

10.4. LECTURA DE DATOS

El proceso a seguir para leer datos de la EEPROM es el siguiente:

1. Escribir la dirección en EEADR.
2. Poner el bit EEPGD de EECON1 a 0 (acceso a EEPROM de datos).
3. Poner a 1 el bit RD (iniciar operación de lectura).
4. Leer el dato en el registro EEDATA.

El dato está listo al ciclo inmediatamente posterior a la puesta a 1 del bit RD, por lo que se puede leer ya en la siguiente instrucción. A continuación se implementa el código necesario para realizar estas acciones:

```
banksel   EEADR           ;Seleccionar Bank2
movfw     DIR             ;dirección donde vamos a escribir
movwf     EEADR           ;Escribir dirección en EEADR
bsf       STATUS,RP0      ;Seleccionar bank3
bcf       EECON1,EEPGD    ;Acceso a EEPROM de datos
bsf       EECON1,RD       ;Iniciar operación de lectura
bcf       STATUS,RP0      ;Volver a bank2
movfw     EEDATA          ;Dato en W
```

10.5. ESCRITURA DE DATOS

El proceso a de escritura es más complejo. El PIC posee varios mecanismos de control para evitar escrituras accidentales, por lo que aunque el proceso de escritura pueda parecer un poco engorroso, es necesario hacerlo así por seguridad (recordar que el número de ciclos de escritura es limitado). Para poder escribir un dato, se deben habilitar las operaciones de escritura mediante el bit WREN. Además, por cada byte que se escribe, como medida de seguridad adicional, se deben realizar una serie de operaciones con EECON2. El proceso completo a seguir sería:

1. Escribir la dirección en EEADR.
2. Poner el bit EEPGD de EECON1 a 0 (acceso a EEPROM de datos).
3. Escribir el dato en EEDATA
4. Poner a 1 el bit WREN para permitir operaciones de escritura.
5. Secuencia: Escribir 55h en EECON2, escribir AAh en EECON. Poner a 1 el bit WR.
6. Deshabilitar escrituras borrando el bit WREN.

El proceso de escritura lleva un tiempo (el fabricante establece un tiempo de escritura típico de 4ms y máximo de 8ms), aunque es independiente del resto del hardware, por lo que una vez que hemos iniciado la operación, podemos seguir con el programa sin problemas. Lo que si es importante, es no acceder a la EEPROM mientras dure este proceso, por lo que en ocasiones será necesario comprobar si ha finalizado correctamente. Esto se puede hacer bien mediante una rutina de retardo, bien comprobando que se ha puesto a 0 el bit WR, o bien si utilizamos interrupciones comprobando el bit EEIF de PIR2.

Con respecto a las interrupciones, es necesario aclarar lo siguiente. Puesto que es crítico que la secuencia de escritura descrita en el paso 5 no se interrumpa, es importante deshabilitar las interrupciones durante este proceso. El siguiente fragmento de código muestra el proceso de escritura:

```
banksel   EEADR          ;Seleccionar Bank2
movfw     dir            ;dirección donde vamos a escribir
movwf     EEADR          ;Escribir dirección en EEADR
movfw     dato
movwf     EEDATA         ;escribir el dato en EEDATA
bsf       STATUS,RP0     ;Seleccionar bank3
bcf       EECON1,EEPGD   ;Acceso a EEPROM de datos
bsf       EECON1,WREN    ;habilitar escrituras
bcf       INTCON,GIE     ;deshabilitar interrupciones
;-------  secuencia de escritura  ------------
movlw     55h
movwf     EECON2         ;Escribir 55h en EECON2
movlw     AAh
movwf     EECON2         ;Escribir AAh en EECON2
bsf       EECON1,WR      ;Iniciar operación de escritura
;---------------------------------------------
bsf       INTCON,GIE     ;Habilitar interrupciones
bcf       EECON1,WREN    ;deshabilitar escrituras
```

Algo muy frecuente, es escribir datos en la memoria EEPROM en tiempo de grabación. Esto se puede hacer de varias formas. Todos los programas de grabación permiten esto. También esto se puede hacer utilizando la directiva **de** definida en el capítulo dedicado al ensamblador MPASM.

11. INTERRUPCIONES

11.1. NECESIDAD DE LAS INTERRUPCIONES

Las interrupciones son un desvío en el flujo normal del programa, al igual que las subrutinas. La diferencia fundamental entre ambas es que éstas son provocadas en el momento en que se da un hecho singular (normalmente relacionado con el hardware) y por tanto **no** podemos predecir el momento en que van a suceder. Las interrupciones resultan una característica muy útil a la hora de trabajar con periféricos.

Supongamos que tenemos un programa que tiene que leer datos de un periférico, por ejemplo el conversor AD. Como sabemos, el conversor necesita de un cierto tiempo para realizar la conversión, por lo que el dato no estará listo hasta que el conversor nos indique que así es. ¿Cómo podemos saber si el dato esta listo? Una posible forma es por encuesta (polling) como ya vimos en el tema del convertidor A/D. Podemos consultar el bit GO/DONE' del registro ADCON0. Si es 0, es que la conversión ha terminado. Esto por supuesto, funciona, pero nos plantea inconvenientes como el hecho de que tenemos que estar constantemente consultando el estado del bit, no pudiendo hacer otra cosa mientras tanto. Otra opción sería dejar al microcontrolador trabajar y consultarlo solo en un momento dado, pero así corremos el riesgo de hacerlo a destiempo y perder muestras.

Este problema es aún más crítico cuando se trata de una aplicación en tiempo real. Supongamos que queremos diseñar una aplicación que usa tiempo real (un reloj con alarma, por ejemplo). Para medir el tiempo usamos uno de los temporizadores del PIC, (el TMR1, por ejemplo). En nuestro programa nos tendremos que encargar de varias tareas, como actualizar la hora, comprobar la alarma, sacar los datos por el display… y comprobar el estado del temporizador para actualizar los datos. Se puede dar el caso más que probable, que al consultar el contador después de haber realizado todas esas tareas, éste ya se haya desbordado hace tiempo y no en el momento exacto en que lo consultemos, con la correspondiente pérdida de precisión que ello conlleva.

Estos ejemplos resultan lo suficientemente ilustrativos de cuan necesario es un método de informarnos de los cambios en un periférico alternativo a la encuesta constante.

El PIC 16F88 tiene múltiples fuentes de interrupción para ayudarnos en esta tarea. Si lo configuramos convenientemente, podemos conseguir que en cuanto se produzca un hito dado (fin de la conversión A/D, contador que se desborda, tecla que se pulsa…) el controlador deje de hacer lo que

está haciendo en ese momento y atienda en el acto a la fuente de la interrupción, consiguiendo así solventar los problemas antes citados. En cuanto se haya atendido la interrupción convenientemente, el controlador podrá retomar la tarea que estaba realizando en el mismo punto donde lo dejó. Hay que tener en cuenta que el programa principal, no tiene que encargarse de consultar el estado del periférico en cuestión, ya que el mismo nos avisará cuando sea necesario, con la simplificación en el código que esto conlleva. Por otra parte, el hecho de que la interrupción se pueda producir en cualquier momento, nos obliga a tomar ciertas precauciones para no perder información, como ya veremos. A continuación vamos a ver que fuentes de interrupción tiene el PIC16F88, así como la forma en que son tratadas.

11.2. Interrupciones en el PIC16F88

Las interrupciones en el PIC se controlan mediante los registros **INTCON, PIR1, PIE1, PIR2 y PIE2**, que veremos más adelante.

El PIC16F88 dispone de varias fuentes de interrupción, todas ellas configurables por separado. Podemos distinguir entre interrupciones externas (producidas en un cambio en una patilla determinada) o internas (producida por alguno de los periféricos integrados). En la tabla siguiente se pueden apreciar las distintas fuentes de interrupción existentes.

Desbordamiento del TMR0
Desbordamiento del TMR1
Coincidencia TMR2 con PR2
Interrupción externa en la patilla RB0/INT
Cambio en las patillas RB4, RB5, RB6 o RB7
Fin de conversión A/D
Buffer de recepción AUSART lleno
Buffer de transmisión AUSART vacío
Fin de recepción/transmisión en módulo SSP
CCP1:Captura en registro TMR1
CCP1: Comparación coincidente en TMR1
Fallo en reloj del sistema
Cambio en entradas del comparador
Ciclo de escritura en EEPROM finalizado

Tabla 22: Fuentes de interrupción

Para poder incluir interrupciones en nuestro programa, debemos hacer lo siguiente:

1. Habilitar interrupciones poniendo el bit GIE de INTCON a 1.
2. Si se trata de un periférico, poner también a 1 el bit PEIE de INTCON.
3. Habilitar la fuente en cuestión poniendo a 1 el bit correspondiente en INTCON, PIE1 o PIE2.

Cuando se produce una interrupción y el bit de habilitación correspondiente está a 1, el procesador salta a la dirección de atención a interrupción que es la 0004h. Esto debe ser tenido en cuenta a la hora de escribir nuestros programas, ya que el vector de reset está en la dirección 0000h, por lo que si nuestro programa comienza en esa dirección y no se desvía, cuando se produzca una interrupción se saltará ahí, donde tendremos código del programa principal. Para evitar esto, debemos desviar nuestro programa a otra dirección como se puede ver en el ejemplo siguiente:

```
ORG     0
    goto inicio     ;saltar a dirección posterior a rutina de INT

    ;vector de interrupciones
    ORG 4           ;aquí se atienden las interrupciones
    …
    …
    retfie          ;fin de rutina de interrupciones

                    ;donde termine la rutina de interrupciones
    inicio          ;debe comenzar el programa principal
```

En la rutina de atención a interrupción, escribiremos el código correspondiente. Debemos tener en cuenta que cuando se produce una interrupción, se salta siempre a la dirección 0004h independientemente de la fuente que la haya provocado. Es labor nuestra hallar la fuente en el caso de que tengamos varias habilitadas, consultando los flags correspondientes en INTCON, PIR1 o PIR2.

Los flags de interrupción se ponen a 1 cuando se produce un determinado hito. Es labor del programador ponerlos a 0. Esto se hace, según la fuente, poniendo el bit a 0 directamente o bien al realizar una determinada operación de lectura o escritura en un determinado registro. Por ejemplo, el bit de interrupción por fin de conversión A/D se desactiva poniéndolo a 0 con una orden BCF, pero el bit de buffer de transmisión AUSART vacío se pondrá a 0 al escribir de nuevo en el buffer.

De la rutina de interrupción se vuelve con la orden RETFIE. Es similar a la orden RETURN, con la salvedad de que RETFIE regresa y además pone el bit de habilitación global de interrupciones (GIE) a

1. Es aconsejable que mientras atendemos una interrupción no se produzca otra, por lo que al producirse una interrupción el bit GIE de INTCON se pone automáticamente a 0. Con RETFIE se vuelve a poner a 1 al salir de la rutina de interrupción.

Puesto que el salto a la rutina de interrupción se puede producir en cualquier fragmento del programa y sin previo aviso, debemos tomar ciertas precauciones. En la rutina de interrupción se modificarán con toda seguridad varios registros que tal vez se estén usando en el programa principal en el preciso instante en que se produce la rutina de interrupción (W, STATUS…). Es labor del programador guardar el contenido de dichos registros para que no se pierda y recuperarlo cuando la rutina de atención a interrupción termine. A continuación se pone un ejemplo de cómo guardar el contenido de W y STATUS. Esto se debe realizar con cualquier registro usado en el programa principal que sea susceptible de ser modificado por la rutina de interrupciones.

```
;rutina de atención a interrupción
ORG 4                           ;aquí se atienden las interrupciones
movwf     copiadeW              ;guardar W
movfw     STATUS                ;y STATUS (en este orden)
movwf     copiadeSTATUS
…
Codigo de atención a interrupción
…
movfw     copiadeSTATUS  ;recuperar el valor de STATUS
movwf     STATUS
movfw     copiadeW              ;y de W (en este orden)
retfie                          ;vuelta al programa principal.
```

Otro factor a tener en cuenta, es el hecho de que si nuestro programa trabaja con registros situados en cualquiera de los bancos de memoria, al saltar a la rutina de interrupción nos encontraremos en el mismo banco en el que estuviésemos anteriormente. Debemos asegurarnos de estar en el banco correcto a la hora de trabajar con registros en la rutina de atención a interrupciones modificando los bits RP1 Y RP0. Evidentemente, esto debe hacerse después de guardar STATUS. Al restaurar STATUS al final de la rutina, volveremos al banco de origen.

A continuación se analizan los registros relacionados con interrupciones.

11.3. INTCON

7	6	5	4	3	2	1	0
GIE	**PEIE**	**TMR0IE**	**INT0IE**	**RBIE**	**TMR0IF**	**INT0IF**	**RBIF**

El registro INTCON está mapeado en todos los bancos de memoria en la dirección 0Bh, contiene los bits de habilitación general de interrupciones, así como los de habilitación y flags de las fuentes de interrupción más usadas.

GIE: Habilitación general de interrupciones. Debemos escribir un 1 si queremos utilizar las interrupciones (cualquiera).

PEIE: Habilitación de interrupciones provocadas por los periféricos. Con 1 se habilitan. Los flags y bits de habilitación se hallan en los registros PIE1, PIR1, PIE2 y PIR2.

TMR0IE: Habilitación de interrupción por overflow en TMR0.

INT0IE: Habilitación de interrupción en la patilla RB0/INT.

RBIE: Habilitación de interrupción por cambio en las patillas RB4, RB5, RB6 o RB7.

TMR0IF: Indicación de desbordamiento del TMR0.

INT0IF: Cambio en la patilla RB0/INT.

RBIF: Cambio en cualquiera de las patillas RB4, RB5, RB6 o RB7. Este bit no se puede poner a cero de nuevo hasta que no se haya leído PORTB.

11.4. PIE1, PIE2

Los registros PIE1 y PIE2 se hallan mapeados en las direcciones 0C y 0D del bank1 (8C, 8D) respectivamente. Contienen los bits de habilitación de las interrupciones producidas por los periféricos. Las interrupciones se habilitan poniendo el bit concreto a 1 y se deshabilitan poniéndolo a 0. **Además es necesario haber habilitado también el bit PEIE** del registro INTCON.

PIE1

7	6	5	4	3	2	1	0
-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

ADIE: Habilitar interrupción por fin de conversión en el convertidor A/D.

RCIE: Habilitar interrupción por recepción en el módulo AUSART.

TXIE: Habilitar interrupción por transmisión en el módulo AUSART.

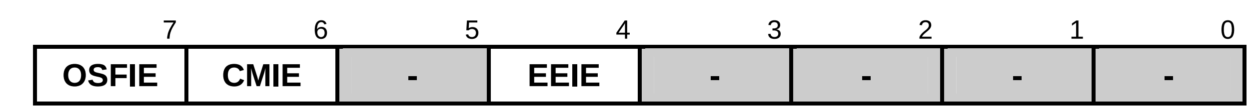
SSPIE: Habilitación de interrupción en puerto serie síncrono (SSP).

CCP1IE: Habilitación de interrupción en el módulo CCP1.

TMR2IE: Habilitar interrupción por coincidencia de TMR2 con PR2.

TMR1IE: Habilitar interrupción por overflow en TMR1.

PIE2



OSFIE: Habilitar interrupción por fallo en el oscilador.

CMIE: Habilitar interrupción por el comparador.

EEIE: Habilitar interrupción por escritura en EEPROM.

11.5. PIR1, PIR2

Los registros PIR1 y PIR2 se hallan mapeados en las direcciones 0C y 0D del bank0 respectivamente. Contienen los flags de las interrupciones producidas por los periféricos y el orden de los mismos es igual que en los registros de habilitación PIE1 y PIE2. Hay que decir que estos flags se activan siempre, tengamos o no habilitadas las interrupciones y deben ponerse a 0 por software una vez que hayamos atendido la interrupción.

PIR1

7	6	5	4	3	2	1	0
-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

ADIF: Fin de conversión en el convertidor A/D (debe ponerse a 0 por software).

RCIF: Buffer de recepción AUSART lleno (se pone a 0 al leer RCREG).

TXIF: Buffer de transmisión AUSART vacío (se pone a 0 al escribir en TXREG).

SSPIF: Fin de recepción/transmisión en SSP (debe ponerse a 0 por software).

CCP1IF: Interrupción en el módulo CCP1.

Captura: captura en registro TMR1 (debe ponerse a 0 por software).

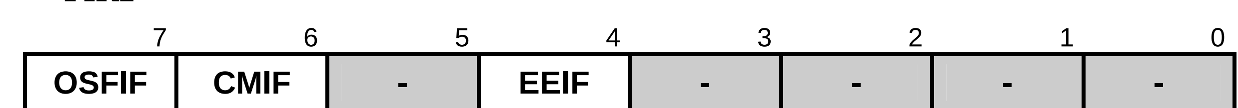
Comparación: coincidencia en TMR1 (debe ponerse a 0 por software).

PWM: No usado

TMR2IF: Coincidencia de TMR2 con PR2 (debe ponerse a 0 por software).

TMR1IF: Overflow en TMR1 (debe ponerse a 0 por software).

PIR2



7	6	5	4	3	2	1	0
OSFIF	CMIF	-	EEIF	-	-	-	-

OSFIF: Fallo en reloj del sistema. Cambio del reloj a INTRC (debe ponerse a 0 por software).

CMIF: Entrada en el comparador ha cambiado (debe ponerse a 0 por software).

EEIF: Operación de escritura en EEPROM finalizada. (debe ponerse a 0 por software).

El siguiente fragmento de código usa interrupciones. Es una modificación del contador de segundos que pusimos como ejemplo para el módulo TMR1. Al usar interrupciones, no tenemos que estar comprobando todo el tiempo el estado del contador, pudiendo dedicarse el programa a otras cosas.

```
ORG  0          ;evitar pasar por rutina de interrupcion
     goto       inicio
ORG 4           ;vector de interrupciones
     movwf      copia_W   ;guardar estado
     movfw      STATUS
     movwf      copia_ST
     bcf        PIR1,TMR1IF         ;deshabilitar flag TMR1

     …               ;codigo de cuenta de segundos
     …

     movfw      copia_ST  ;recuperar estado
     movwf      STATUS
     movfw      copia_W
     retfie

inicio
     bsf        STATUS,RP0     ;seleccionar bank1
     bsf        OSCCON,IRCF1   ;oscilador a 250KHz (IRCF(2:0)=010
     bsf        INTCON,GIE     ;activar interrupciones
     bsf        INTCON,PEIE    ;interrupciones perifericos activadas
     bsf        PIE1,TMR1IE    ;activar interrupcion para TMR1
     clrf       TRISA
     clrf       TRISB
     bcf        STATUS,RP0     ;seleccionar bank0

bucle
     …          ;acciones a realizar en el programa principal
     …
     goto       bucle
     END
```

12. EL ENSAMBLADOR MPASM™

La programación del PIC 16F88 se puede hacer en varios lenguajes. Tenemos disponibles compiladores de C, BASIC y otros lenguajes de alto nivel, que constituyen la mejor alternativa a la hora de realizar programas de cierta complejidad. No obstante, a la hora de conocer el hardware del sistema para el cual estamos desarrollando aplicaciones, el lenguaje que más cumple con este propósito en el propio ensamblador, ya que es más parecido a diseñar un circuito que a escribir software. Otra ventaja del ensamblador, es que su aprendizaje puede resultar más sencillo que el de un lenguaje de alto nivel. Ya hemos visto varias características del ensamblador del PIC, como el juego de instrucciones, la sintaxis, los mnemónicos…

Lo que vamos a ver en éste capítulo es un estudio más completo del ensamblador MPASM™ de la firma Microchip para poder realizar ficheros fuentes completos, o sea, ficheros ya preparados para ser compilados y grabados en un microcontrolador.

12.1. SINTAXIS GENERAL

Etiquetas

El uso de etiquetas es muy importante para marcar direcciones de memoria y conseguir un código más legible. Las etiquetas pueden ser cualquier palabra, pero con ciertas restricciones: no pueden empezar con dos guiones bajos (`__INICIO`), ni con un guión bajo seguido de un número (`_2bucle`), ni ser una palabra reservada (`goto`). Las etiquetas pueden terminar con dos puntos, aunque no es necesario. Es importante resaltar que las etiquetas distinguen las mayúsculas, por lo que no es lo mismo la etiqueta *Inicio* que *inicio*. Cuidado con esto a la hora de escribir programas.

Se recomienda situar las etiquetas en la columna 1, para dar una mayor legibilidad al código. A continuación se ve un ejemplo con etiquetas:

```
bucle  MOVFW    PORTA
       MOVWF    PORTB
       GOTO     bucle
```

Mnemónicos

El MPASM™ incluye mnemónicos para las instrucciones, pero también para los registros y los bits específicos de cada registro. Por ejemplo, para poner a 1 el bit RP0 del registro STATUS (Se recuerda que este bit es el 5 y el registro STATUS está mapeado en todos los bancos en la dirección

03h), no tendríamos que escribir `BSF 0x03,5` si no que escribiríamos directamente: `BSF STATUS,RP0`. Esto hace que la programación sea más sencilla al no tener que trabajar con direcciones de memoria y el ordinal del bit. Los mnemónicos se pueden poner en mayúsculas o minúsculas.

Literales

Los valores numéricos que queramos cargar en **W**, se pueden expresar de distinta forma. A continuación se pueden ver las distintas formas de expresar los literales:

Decimal:	`MOVWF`	`d’118’ o .118`
Octal:	`MOVWF`	`o’118’`
Hexadecimal	`MOVWF`	`0x76 o h’76’`
Binario	`MOVWF`	`b’01110110’`
ASCII:	`MOVWF`	`a’Z’ o ’Z’`

Comentarios

En todos los lenguajes de programación se hace necesario usar comentarios. Los comentarios son un texto incluido por el programador para explicar algo sobre el programa. Es muy importante que nos acostumbremos a incluir gran cantidad de comentarios en nuestros programas por varias razones: En primer lugar, porque son imprescindibles para comprender el código. Recordemos que el lenguaje ensamblador puede resultar bastante farragoso, por lo que puede pasar que con la sola lectura del código no sepamos que es lo que hace el programa. Incluir comentarios que expliquen que hace ese fragmento de código concreto, ayudará a comprenderlo con más facilidad a quien lea el código, ya sean otros programadores o nosotros mismos (aunque lo tengamos muy claro y lo veamos muy fácil y obvio en el momento de escribir el código, seguro que no lo vemos igual al cabo de unos días). En segundo lugar, los comentarios son “gratis”. Son líneas que se incluyen en el fichero fuente, pero que en el momento de compilar el código son eliminadas por el compilador, por lo cual, en el código máquina no van a aparecer, solo las veremos en el fichero fuente. Los comentarios comienzan por ‘**;**’ y ocupan una línea. Sí necesitamos más líneas, cada una de ellas deberá comenzar por ‘**;**’ también.

Lo que se ha explicado sobre los comentarios, se puede comprobar a la perfección intentando comprender los siguientes fragmentos de código:

Sin comentarios:

```
MOVFW       PORTA
MOVWF       PORTB
RLF         PORTB
RLF         PORTB
```

```
ADDWF        PORTB
```

Con comentarios:

```
; Este programa lee el dato en el puerto A
; lo multiplica por 5 y lo saca por el puerto B
MOVFW        PORTA          ; copiamos PORTA a W
MOVWF        PORTB          ; y W a PORTB
RLF          PORTB          ; multiplicamos Nx2 (despl. A izq)
RLF          PORTB          ; 2 veces (Nx4)
ADDWF        PORTB          ; y sumamos el numero ( Nx4 + N = Nx5)
```

12.2. DIRECTIVAS

Las directivas son comandos que aparecen en el código fuente pero no son códigos de operación del microcontrolador. Se usan para controlar el ensamblado, entrada, salida, localización, simplificar el código… Vamos a ver un número limitado de directivas, solo aquellas que utilicemos con más frecuencia en nuestros programas. Para conocer más leer la guía de usuario de MPASM™.

banksel

Genera el código para selección de banco. Resulta útil para no tener que escribir el código para cambiar los bits RP0 y RP1 cada vez que tenemos que cambiar de banco. Escribir:

```
BANKSEL      TRISA     ; seleccionar banco de TRISA (banco 1)
```

Es lo mismo que escribir:

```
BSF     STATUS, RP0    ; seleccionar banco 1
BCF     STATUS, RP1    ; RP0 = 1, RP1 = 0
```

La principal ventaja de la directiva BANKSEL, es el hecho de que se puede apreciar el motivo del cambio de banco en la misma expresión. Al escribir tanto BANKSEL ADRESL como BANKSEL ANSEL hacemos exactamente lo mismo: vamos a bank1, pero al escribir siempre el mnemónico del registro, sabemos el porqué del cambio de banco.

cblock, endc

Genera un bloque de constantes en una dirección de memoria. Se utiliza para definir variables en nuestro programa. El siguiente ejemplo muestra como utilizar esta directiva:

```
cblock 0x20    ;comenzamos en la dirección 20h
   Temp_ext    :2
   Temp_int    :2
endc
…
…
cblock         ;comenzamos en la dirección 24h
   segundos
   minutos
   horas
endc
```

Se puede especificar la cantidad de posiciones que reservamos para cada variable. En el ejemplo anterior las variables de temperatura ocupa dos posiciones. Si no ponemos nada, se reserva una única posición. Después de CBLOCK se indica la dirección de memoria a partir de la cual se reserva la memoria. Si no ponemos nada, se reservará a partir de la siguiente al CBLOCK anterior. Esto es útil a la hora de reservar memoria en ficheros externos sin necesidad de definir donde está, ya que el compilador se encargará de asignarlas la siguiente posición libre independientemente de su valor.

`__config`

Define los bits de configuración del procesador. Si no lo ponemos, se usa una configuración preasignada, que no siempre será la que necesitemos. De hecho, muchas veces el oscilador interno es más que suficiente para nuestras aplicaciones y por defecto el micro se configura para usar uno externo. El PIC 16F88 tiene dos palabras de configuración que se identifican como `_CONFIG1` y `_CONFIG2`. Aunque nosotros solo usaremos `_CONFIG1`, tenemos que indicar que nos referimos a ésta poniéndolo a continuación de `__CONFIG`.

Los bits de configuración se pueden incluir de dos maneras: Bien poniendo la característica que queremos activar o desactivar mediante sus mnemónicos y uniéndolas con &, o bien poniendo el valor hexadecimal de esa configuración, que vimos en el apartado 5.7. Una forma de obtener este valor fácilmente, es escribirlo en el software de grabación ICPROG que veremos en un tema posterior. La palabra de configuración es lo primero que se incluye en el código justo después de la directiva `include` que veremos más adelante. A continuación se incluyen dos formas de escribir la misma palabra de configuración, es la misma que vimos en el ejemplo del apartado 5.7.:

```
__CONFIG    _CONFIG1, _CP_ALL & _CCP1_RB0 & _DEBUG_OFF & _CPD_OFF
& _WRT_PROTECT_OFF & _LVP_OFF & _BODEN_OFF & _MCLR_OFF & _PWRTE_ON
& _WDT_OFF & _INTRC_IO

__CONFIG    _CONFIG1, (3f10)
```

de

Define EEPROM. Genera una serie de valores para ser escritos en la EEPROM del PIC en tiempo de grabación (Constantes, Contraseñas…). Además del valor, debemos poner la dirección de la EEPROM donde estará dicho valor, teniendo en cuenta que en el MPASM, la EEPROM se encuentra mapeada a partir de la dirección 2100h. El siguiente ejemplo guarda una contraseña de 4 números en la EEPROM a partir de la dirección 00h de la misma:

```
ORG     0x2100          ;comenzar en la dirección 00h de la EEPROM
de      clave  1,2,3,4 ;1 en 00h, 2 en 01h…
ORG     0               ;no olvidar volver a la memoria de programa
Programa    …
```

dt

Define Table. Genera una serie de instrucciones `retlw`, cada una de ellas con un valor de 8 bits. Es muy útil a la hora de definir tablas de datos en la memoria de programa. Para acceder a un dato de una tabla, situamos en `W` el índice del elemento al que queremos acceder, llamamos a la dirección de la tabla con la instrucción `call`, y sumamos `W` a `PCL`. El PC saltará a esa instrucción `retlw` concreta y se regresará de la subrutina con el dato en W. En el siguiente ejemplo se ve un uso de la directiva dt:

```
; representar el número introducido en A3..A0 en un display
; de 7 segmentos conectado en B6..B0
bucle   movfw     PORTA          ; W = PORTA
        call      7seg           ; ir a 7seg
        movwf     PORTB          ;sacar por PORTB el valor devueto
        goto      bucle
7seg:   addwf     PCL            ; sumar W a PCL (o sea, PORTA)
        dt        b'00111111',b'00000110',b'01011011',b'01001111'
        dt        b'01100110',b’01101101',b'01111100',b'00000111'
        dt        b'01111111',b'01100111',b'01110111',b'01111100'
        dt        b'00111001',b'01011110',b'01111001',b'01110001'
```

La directiva `dt` del código anterior se traducirá por las instrucciones:

```
retlw     b'00111111'    ; segmentos a encender para el 0
retlw     b'00000110'    ; el 1
retlw     b'01011011'    ; el 2
retlw     b'01001111'    ; el 3, etc
..
..
```

El manejo de tablas en el PIC puede ser más complejo de lo que a priori aparenta. Recordemos que el PC está dividido en dos registros de 8 bits, por lo que si en algún punto de la tabla se desborda PCL al sumar el Offset, al no actualizarse también el PCLATH, tendremos un error y accederemos a una posición de memoria distinta. Es importante que esto no suceda. Para evitar esto, puede ser conveniente colocar las tablas al comienzo de la memoria de programa:

```
ORG     0
  goto programa
  ;tablas al comienzo de memoria. ojo si hay interrupciones!!!
Pide_Clave
  addwf     PCL
  dt        “Introduzca codigo” ,0
Clave_Correcta
  addwf     PCL
  dt        “Codigo Correcto!”,0
  …
programa
  …
```

Además de esto, el PIC no trabaja bien con tablas situadas por encima de las primeras 256 posiciones de memoria. Y hay ocasiones en las que esto será inevitable. Para poder solucionar esto, se debe modificar el registro PCATLH de la forma siguiente:

```
Tabla_25
  movwf   guardaPCL          ;guardar W (offset de la tabla)
  movlw   HIGH Tabla_25      ;meter manualmente en PCLATH
  movwf   PCLATH             ;el valor que corresponde
  movfw   guardaPCL          ;restaurar offset en W
```

```
addwf   PCL                ;y seguir como siempre
dt …
```

Las directivas de ensamblado HIGH y LOW devuelven la parte alta y baja respectivamente de un dato que ocupa más de 8 bits. En este caso, la parte superior de la dirección donde se haya Tabla_25, que es el valor que se debe escribir en PCLATH.

end

Indica el final del programa. Todos los programas que escribamos deberán terminar con esta instrucción. Nada más.

equ

Define una constante de ensamblado. A la hora de ensamblar el programa, todas las referencias a ésta constante se sustituirán por su valor. Es útil para no tener que trabajar con números que en ciertos fragmentos de programa no podemos saber que significan. Se pone antes del comienzo de programa:

```
seg  equ  0x5A      ;nos referiremos a la posición 5A como seg
          ..
     incf seg       ; es lo mismo que poner incf 0x5A
```

#include

Incluye un fichero fuente adicional. El efecto de la directiva `#include “display.h”` sería el mismo que poner el contenido del fichero display.h en el mismo sitio donde hemos puesto el `#include`. Se pueden usar tres tipos de sintaxis:

```
#include fichero
#include “fichero”
#include <fichero>
```

La almohadilla ‘#’ del principio se puede obviar, pero es recomendable ponerla.

Esta directiva la usaremos en todos nuestros programas al principio con la sintaxis: `#include “p16f88.inc”`. En este fichero están definidas varias definiciones correspondientes a nuestro controlador. A continuación se ve un fragmento del contenido de este fichero:

```
..
```

```
PCL                            EQU     H'0002'
STATUS                         EQU     H'0003'
FSR                            EQU     H'0004'
PORTA                          EQU     H'0005'
PORTB                          EQU     H'0006'
..
```

list

Son una serie de instrucciones para el compilador. Debe ir en la primera línea del código fuente. Se usa para varia cosas, pero nosotros solo lo usaremos para poner el tipo de microcontrolador (obligatorio) y también para especificar el radix. El radix, es el formato por defecto de los números, si ponemos, por ejemplo, el radix decimal, hace que todos los números que pongamos sin especificar su base se consideren decimales. Por defecto es hexadecimal.

```
list p=16f88, R  = DEC; programa para pic16f88, radix decimal
..
movlw   70   ; el 70 es decimal
```

org

Indica la posición de memoria donde va a ir localizado nuestro programa. Si no ponemos nada, el programa se ubicará en la dirección 0000h. Muchas veces esto no es conveniente, por ejemplo si usamos interrupciones, ya que como vimos, el vector de reset está en la dirección 0000h y el de interrupciones en la 0004h, por lo que si se produce una interrupción, se salta a esa dirección, en la que tiene que estar la rutina de servicio de interrupción. En el siguiente fragmento de código se puede ver como organizar un programa con interrupciones:

```
org 0       ; este código se ubica en la dirección 0000h
goto Inicio ; saltar a Inicio
org 4       ; este código va en 0004h (rutina de interrupción)
goto Int    ; saltar a Int (lugar donde estára la rutina de int)
..
org 5       ; Este código se ubica en 0005h
Inicio      ; aquí comienza el programa
..
..
org 100     ; Este código se ubicará en 0100h
Int         ; aquí está la rutina de interrupción
```

```
..
..
END         ; el programa termina con END
```

res

Reserva memoria para datos. Es muy similar a `equ`, con la diferencia de que con res no especificamos la dirección (el compilador las va ubicando automáticamente) y podemos reservar varias unidades de memoria. Esto se ve en el siguiente ejemplo:

```
ORG 0x20  ;hacer que las variables comiencen en la dirección 20h
  Contador        res  1    ;variable en 0x20
  Codigo          res  4    ;4 posiciones en 0x21 .. 0x24
  N_Intentos      res  1    ;variable en 0x25
ORG  0
programa
…
```

Cuando utilizamos varios ficheros de código con #include, cada uno con sus propias variables, es importante actualizar el puntero de dirección de variables para que esto se haga de forma correcta. El siguiente ejemplo muestra como utilizar correctamente la directiva **res** en más de un fichero:

En el programa principal pondríamos:

```
VARIABLE variables = 0x20    ;variables comienzan en 20h
ORG  variables
  Contador        res  1    ;variable en 0x20
  Codigo          res  4    ;4 posiciones en 0x21 .. 0x24
  N_Intentos      res  1    ;variable en 0x25
variables = $  ;actualizar el puntero de variables
ORG  0
  programa
  …
```

Este fragmento de código necesita varias aclaraciones. En primer lugar, hemos utilizado la directiva **VARIABLE**. Esta directiva, crea una variable de ensamblado, que en este caso hemos llamado *variables*, que almacena un valor que podrá ser alterado posteriormente. En este caso, iniciamos variables a 20h, que si recordamos, es la primera dirección del PIC para registros de propósito general. En la línea siguiente al poner `ORG variables`, hacemos que las directivas res comiencen a reservar memoria a partir de esa dirección. Al finalizar, actualizamos *variables* al último

valor. **$** es una palabra reservada del compilador que es el puntero a la dirección actual. En este caso **$** toma el valor 25h, que es al que se actualizará *variables*.

En los ficheros con funciones externas definiríamos las variables de la forma que viene a continuación. Es importante reseñar que la invocación a éste fichero mediante la directiva **#include**, debe hacerse después del código anterior. Lo normal es poner estas llamadas al final del programa principal justo antes de la directiva **END**, pero conviene recordarlo.

```
ProgrRetardos  equ $
ORG  variables
   R_ContA     res  1           ; Contadores para los retardos.
   R_ContB     res  1
   R_ContC     res  1
variables = $
org  ProgrRetardos
    …
```

Guardamos la posición en la que se encuentra el programa en *ProgRetardos*. Esto es importante, ya que al poner `ORG variables`, lo que viene a continuación se situará en el valor de variables (25h) y perderemos dicha dirección. Reservamos las posiciones necesarias igual que antes y al final actualizamos nuevamente el valor de *variables* y mediante la directiva `ORG Progretardos` hacemos que el código de este fichero se sitúe a continuación del anterior. En cada fichero que necesitemos incluir, usaremos esta sintaxis para definir las variables.

13. Desarrollo y SIMULACIÓN: PROTEUS™

Vamos a diseñar un contador Gray de 3 bits con biestables D. Seguiremos el mismo procedimiento que en el apartado anterior.

a tenemos el dado, el problema resultará más sencillo.

14. GRABACIÓN: ICPROG™

Vamos a diseñar un contador Gray de 3 bits con biestables D. Seguiremos el mismo procedimiento que en el apartado anterior.

a tenemos el dado, el problema resultará más sencillo.